ESD-TR-70-297

STUDIES IN EXTENSIBLE PROGRAMMING LANGUAGES

Ben Wegbreit

May 1970

DIRECTORATE OF SYSTEMS DESIGN AND DEVELOPMENT
HQ ELECTRONIC SYSTEMS DIVISION (AFSC)
L. G. Hanscom Field, Bedford, Massachusetts 01730

(Prepared under Contract No. F19628-68-C-0101 by Harvard University, Cambridge, Massachusetts.)

LEGAL NOTICE

When U. S. Government drawings, specifications or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

OTHER NOTICES

Do not return this copy. Retain or destroy.

ESD-TR-70-297

STUDIES IN EXTENSIBLE PROGRAMMING LANGUAGES

Ben Wegbreit

May 1970

DIRECTORATE OF SYSTEMS DESIGN AND DEVELOPMENT
HQ ELECTRONIC SYSTEMS DIVISION (AFSC)
L. G. Hanscom Field, Bedford, Massachusetts 01730

(Prepared under Contract No. F19628-68-C-0101 by Harvard University, Cambridge, Massachusetts.)

FOREWORD

This report is based on a thesis submitted on May 30, 1970 by Ben Wegbreit to The Division of Engineering and Applied Physics, Harvard University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This report was prepared in support of Project 2801, Task 280102 by Harvard University, Cambridge, Massachusetts under Contract F19628-68-C-0101, monitored by Dr. John B. Goodenough, ESMDA, and was submitted 17 August 1970.

This technical report has been reviewed and is approved.

Sylvia R. Mayer
SYLVIA R. MAYER
Project Officer

William F. Heisler
WILLIAM F. HEISLER, Colonel, USAF
Director of Systems Design & Development
Deputy for Command & Management Systems

ABSTRACT

This work is a study of two topics in the development of an extensible programming language, i.e., a high level language with powerful definitional facilities so designed that the language can be extended and thereby tailored for use in a wide variety of computer applications. The first topic is a theoretical treatment of an extension facility for syntax. It generalizes the notion of context-free grammars to allow the syntax of a language to be a function of its generated strings. It studies the formal properties of such grammars and presents an efficient algorithm for parsing their languages. The second topic of this work is a study of the design and formal specification of a base language on which an extensible language system can be built. It employs a formal definition to present a base language, examines the constraints on the design of such language, and discusses how these constraints shape the language. The language includes one extension facility, that for data types; the facility, its design, and its relation to similar facilities in other languages are analyzed.

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (concluded)

. . . a good notation has a subtlety and suggestiveness which at times make it seem almost like a live teacher . . . a perfect notation would be a substitute for thought.

Bertrand Russell
in the introduction to Wittgenstein's
Tractatus Logico-Philosophicus

Chapter 1

INTRODUCTION

High-level problem-oriented programming languages were proposed to reduce the time and cost of programming by enabling the programmer to specify procedures in a concise language appropriate to some problem area (e.g., cf. [Back57]). Initially, this was simple enough: there were numerical scientific problems and there were business data processing problems. In time, however, the set of application areas grew larger; a list would now include discrete simulation, algebraic manipulation, artificial intelligence, string and text processing, machine tool control, civil engineering, information retrieval, and computer graphics. There is no reason to believe that the growth in areas of computer applications has come to an end. Further, specific problems frequently fail to fall neatly into a single application area. It is sometimes necessary to perform algebraic manipulation followed by numerical calculation, discrete simulations with results displayed graphically, or business data processing coupled with text processing and report preparation. In the future, such sprawl of problems over several application areas will increase and may even become the rule.

Traditionally, computer science has attempted to provide one or more languages for each application area. This is expensive. Expensive in language design, implementation, and maintenance; expensive in programmer training; expensive in system overhead. Further, this leaves unsatisfied the project or programmer whose problem involves more than one of the recognized areas.

More recently, the solution has been to provide a language which serves several applications. For example, PL/I attempts "to encompass among its users the scientific, commercial, real-time, and systems programmers" [Rad65]. CPL was developed with similar objectives [Barr63]. There are two difficulties with this approach.

(1) Such languages are large. To quote from a tutorial paper on PL/I by D. Beech of IBM: "Perhaps the most immediately striking attribute of PL/I is its bulk" [Beech70]. The bulk is hardly surprising, for such languages are essentially created by agglutination of facilities for the several intended application areas. While not surprising it is, however, expensive: in language design, implementation, and maintenance, in programmer training, and in system overhead.

(2) Only a limited, fixed set of application areas is provided for. The programmer who requires significant use of an area not explicitly included in the package is no better off than before. For example, PL/I provides character strings as a data type and has builtin certain simple operations on strings. However, if it is necessary to carry out a pattern-matching and replacement algorithm, e.g. as in Snobol [Garb66], PL/I provides little help. There is no notation other than procedure calls in which to express patterns, so that representation is very clumsy; storage management is awkward; in general, the language serves as a poor host.

Extrapolating into the future, one might expect the next generation of conglomerate languages to provide for combinations such as scientific calculation, data processing, string manipulation, and discrete simulation. Machine tool control and information retrieval might be added in the generation after that, with no end in sight. On the other hand, the continued proliferation of new languages, one or more for each application area, is no better.

The solution is to be found within the milieu of programming. Traditionally, programmers have chosen notation and structure to suppress the constant and display the variable. Subroutine calls, iteration loops, recursion, data description units, and indirect references to data and control are all devices to collect invariants while simultaneously exhibiting the points of variability. Programming languages, at least good programming languages, are designed to give concrete realization to these representation schema. In viewing the flood of application areas, it is clear that the application area is a legitimate variable. Hence, we need a programming language which is itself variable over a comparable range. That is, we should like a language which can be extended, modified, and thereby tailored for use in a wide variety of application areas.

What is constant in such a language is the ability to change. That is, various language facilities are required to allow variation: to accept the definition of extensions and act on them to produce a modified language. These facilities constitute the core of a variable or extensible language and it is precisely these constants which must be provided in the language.

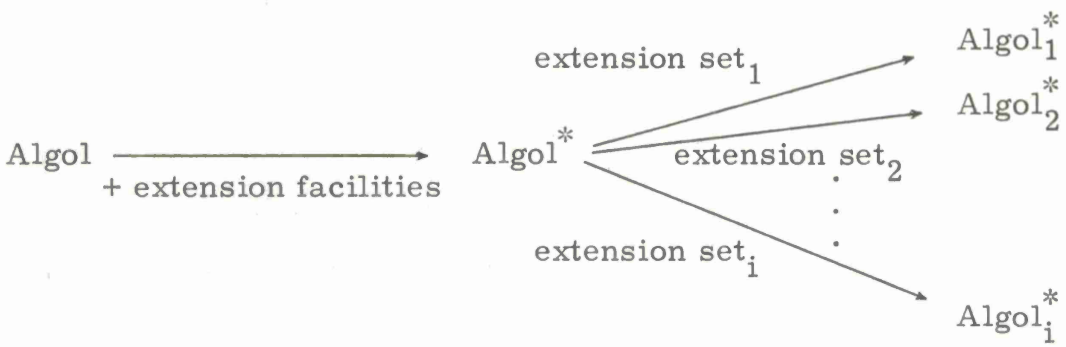
In one sense, Algol 60 is a start at such a language. The numerous proposals of the form: "An Extension to Algol for $\mathscr{X}$" for

$\mathscr{X} \in \{$string manipulation [Smith60], formula manipulation [Perlis66], discrete simulation [Dahl66], synchronous systems [Parn66], . . . $\}$

testify to the mutability of Algol 60 and the durability of the Algol strain under mutation. However, in each case, the authors of the extension were required to go outside of the Algol language, indeed outside of their Algol system, to define and implement the extension. The definition was an English-language report or paper; the implementation required re-writing the compiler. Further, each of these projects was undertaken

separately: there was little realization of the commonality of all projects in extending Algol. Hence, there was little done to find unifying principles or mechanisms which would aid in the $n+1^{st}$ extension.

To outline the envisioned scenario and establish some notation, it will be useful to investigate how a unified extension schema could have been produced. That is, suppose these extensions were to be realized in a unified manner from within the language; what would be required to carry this out? We might accept Algol 60 as defined in the revised report [Naur63] as a base language from which to start our construction. Since Algol 60 does not contain extension mechanisms† these must be added by rewriting the compiler and issuing a new report: "An Extension to Algol for Creating Further Extensions." This language – call it Algol* – differs from the other augments mentioned above in that it is expected to contain them all, in posse. Hence, it is said to be a language core. That is, a core language consists of a base language plus extension facilities. Using the extension facilities, one can define a variety of extension sets, each set creating a new extended language. Pictorially,



The key point is that the extension sets are legal forms in Algol*.

---

†The sole exception is the fluent but weak device of procedure call.

There is little new in this proposal. As early as 1960, J. Smith observed [Smith60] that languages properly belong to language-systems containing a "nested 'continua' of languages. In such systems, new languages may be embedded, appended, extracted at will." The idea was periodically rediscovered. For example, at the Symposium of the International Computation Center, Rome 1962, van der Poel [vanD62] proposed that "what is needed is an extremely powerful and generalized language, but stripped down to the utmost, stripped down to the possibilities that it can, in itself, by a sort of procedure declaration, declare the rest of the mechanism needed." Later in the conference, C. Strachey [Strac62] specifically observed that "it is absolutely essential that this general language should have the facilities of using new syntactic forms."

For a number of years, the idea remained an orphan: born but not adopted. Around 1966 a number of papers appeared, proposing either core languages or particular extension mechanisms which could be grafted onto existing base languages. For the most part, these were paper designs — unimplemented or only partly implemented. However, a brief review of this early work will serve to illustrate the sort of extension facilities envisioned for an extensible language.†

Leavenworth [Leav66] discusses application of the macro concept, familiar in assembly languages, to high-level languages. Taking as illustrative base language a subset of Algol 60, he proposes that macros be used to extend the possible forms for two syntactic types: ⟨ statement ⟩ and ⟨ primary ⟩. For example, a macro which defines a simple type of

---

†In section 2.2 of chapter 3, we discuss in detail more recent proposals for extensible languages.

⟨for statement⟩ may be specified by the macro pattern[†]

⟨statement⟩ ::= for ⟨variable⟩ ←
⟨expression⟩ to ⟨expression⟩ do ⟨statement⟩

where ⟨variable⟩, ⟨expression⟩, and ⟨statement⟩ are syntactic types defined by the base (or extended) syntax. Whenever this pattern is found in the source text during parsing, the matched substring is deleted and replaced by an expansion of the macro definition which is

```
begin $1 ← $2;
    L1: if $1 ≤ $3 then begin $4; $1 ← $1+1; goto L1 end;
end
```

The expansion is performed by replacing each instance of $i by the substring which matches the i^{th} syntactic unit in the pattern. The expanded string is then reparsed, so that multiple levels of definition can be cleanly, if not efficiently, handled.

Cheatham [Chea66] proposes a system which generalizes this in three areas.

(1) Macros may be called either during syntactic analysis, as proposed by Leavenworth, or subsequent to analysis. In the latter case, expansion corresponding to multiple levels of definition can be carried out once, at definition time, rather than on each invocation of the macro.

(2) Also, post-analysis macros may have their meaning expressed in the intermediate language of the translator, giving additional flexibility and control over the semantic definition.

[†]We have taken some liberties in changing the notation used by Leavenworth to be closer to that of Algol 60.

(3) Finally, macros may be defined to be of any syntactic type in the language, not just ⟨primary⟩ and ⟨statement⟩ as proposed by Leavenworth.

Garwick [Gar67] discusses the definition of new data types and operators which act on objects of these new types. The core language has only three types: real, integer, and byte. However, the data type complex may be defined by the declaration

```
block complex {real re , im}
```

Here, "block" signals a certain class of data type declaration, "complex" is the name of the new data type which is defined to consist of two reals, the first being named "re", the second being named "im". Subsequent to definition of complex, variables may be declared to have that data type

```
complex w, z
```

The components of w and z are reals and may be referenced by subscripts which use the component names. For example,†

```
w[re] := z[im]
```

sets the "re" component of w equal to the value of the "im" component of z. Since it is awkward to explicitly denote all operations on complex numbers in terms of their underlying structure, the various arithmetic operators may be extended to operate on complex quantities as well as reals and integers. For example, the assignment operator is extended by the following definition.

---

†Again, we have changed notation, bringing it closer to Algol 60. We thereby avoid explaining several idiosyncrasies of Garwick's notation which are irrelevant to the present discussion.

```
operator  a := b defined
    if (real a ∨ integer a) ∧ complex b
        then a := b[re]
    else if complex a ∧ (real b ∨ integer b)
        then begin a[re] := b; a[im] := 0 end
    else if complex a ∧ complex b
        then begin a[re] := b[re]; a[im] := b[im] end
```

Here, ":=" is the operator being defined; ":=" takes two actual operands which are denoted in the definition by the formal operands "a" and "b". The definition tests the types of the operands using real, integer, and complex as predicates and selects the appropriate defining body. (Note that the defining bodies use ":=" as defined prior to the extension.) In similar fashion, Garwick exhibits definitions of the data types string (character string), vector3 (3-space vectors), and poly (polynomials with real coefficients) and constructs appropriate operations over these types.

Galler and Perlis [Gall67] take a further step. One observes that operations on new data types are defined in terms of operations on their components. Hence, to perform operations on instances of some defined type, e.g. matrix, it is frequently necessary to sequence over their components. For example, if A, B, and C are n by n matrices,

A := B × C

should have the result

```
begin integer i, j, k;
    for i := 1 step 1 until n do
        for j := 1 step 1 until n do
            A[i, j] := innerproduct (B[i, k], C[k, j], k, n)
end
```

where innerproduct is defined in the usual fashion using yet another iteration.† Using the method of Garwick, the matrix statement "A := B × C" would be interpreted as follows.

(1) The operator "×" is called. Since the operands are matrices, the matrix code is selected (at compile-time). Execution of this code produces a result matrix R defined R ≡ B × C.

(2) The operator ":=" is called, the result being that R is assigned to A.

The point is that each operation is performed orthogonally to all others. While this simplifies the processing, the generation of temporaries such as R is logically unnecessary and wastes storage.

The paper by Galler and Perlis is concerned principally with a method for automatically generating expansions such as that given above. The method consists of two steps.

(1) Replacement of operator/operand units by their definition using a scheme much like the syntactic macros of Leavenworth and Cheatham, the difference being that a parse tree rather than string text is used as the program representation.

---

†For example, using the Jensen device [Ruti67] in a type procedure, a possible definition is

```
real procedure innerproduct(x, y, k, n);
    value n;
    real x, y; integer k, n;
    begin real sum;
          sum := 0;
          for k := 1step1 until n do  sum := sum + x × y;
          innerproduct := sum;
    end innerproduct
```

(2) Rearrangement of the resulting parse tree in an attempt to optimize the amount of temporary storage to be allotted. This is carried out by a top-down search of the parse tree, applying a set of transformation rules[†] repeatedly, according to certain cyclical orderings.

While most proposals for extension facilities have been devoted to one or more of the three areas discussed above – syntax, data types, and operators – various other areas have been put forth as candidates for variability (e.g., cf. [Perlis66] and [Stand69]). Most important of these is control, i.e., allowing definition of control structures such as co-routines, pseudo-parallel processes, clock-driven simulation, back-tracking, and monitoring with interrupt capabilities. Related to this is the notion of specifying evaluation rules of special forms, possibly in terms of special control structures.[‡]

In addition to those papers cited, a host of others has appeared in recent years.[§] Despite an abundance of research, a satisfactory extensible language has yet to be designed. That is, while there are languages, even working languages, which exhibit one or more extension mechanisms, no language handles extension with the same completeness and success as, for example, Algol 60 handles the expression of numerical algorithms.

[†] The analogy between this and the transformation rules on deep structure hypothesized by linguists [Chom65] invites investigation. While such investigation lies outside the scope of this work, we feel it may be quite significant and intend to pursue it elsewhere.

[‡] For example, one might introduce the notion of a parallel case expression in which the selection produces a set of integers (instead of the normal integer) in which case, all the corresponding statements are evaluated in parallel, producing as result a list of the individual results.

[§] In chapter 3, section 2.2, we give a reasonably complete list of extensible language projects.

In many respects, the field is in the same state as was the field of languages for numerical algorithms around 1954, 1955. There are languages which can claim the appelation "extensible", but the claim is weak and it is clear that these languages are but first steps. To quote again from Strachey [Strac62], "This proposal [an extensible language] is not one that can be laid down in advance, which can be worked out by an international committee, is is really a difficult problem."

It is our contention that even now, a really satisfactory extensible language is several years away. Much research remains to be done in each of the three principal components which constitute an extensible language:

(1) the base language, its theoretical foundations, its design, and its specification,

(2) the extension facilities, their mechanism, and their theory,

(3) the definition sets, their creation and interaction.

While some of this work can be carried out in conjunction with a language development and implementation project, other topics may be pursued in purely theoretical investigations.

This thesis is a study of two specific issues in extensible languages: (1) formal syntactic specification, and (2) design and formal specification of a base language. The thesis does not attempt design of a complete extensible language and system, nor does it attempt to integrate the two issues into a partial extensible language. The issues are largely orthogonal and we have chosen to treat them independently. Consequently, the two studies are autonomous; each is presented in a separate, self-contained paper.

In the formal syntactic specification of an extensible programming language, the salient characteristic is that the language must be allowed to grow. Assuming that the language is specified by a context-free grammar, this is equivalent to requiring that the grammar be permitted to grow. Additions to the grammar are derived from extensions statements made in the language. For example, the macro pattern of Leavenworth,

⟨statement⟩ ::= for ⟨variable⟩ ← ⟨expression⟩ to ⟨expression⟩ do ⟨statement⟩

which is a declaration permitted in a ⟨blockhead⟩, may be interpreted as a new production to be added to the grammar. To generalize, one might consider a class of grammars in which the production rules used in the derivation of a string are determined in part by the string itself. Note that this requires that the initial grammar must allow the generation of strings which contain substrings interpretable as productions.

In chapter 2, we study the theory and application of such grammars, which we term extensible context-free. We first examine formal properties such as structural results, closure, undecidability results, restricted cases, and relation to other models in formal language theory. Subsequently, we discuss the parsing of languages specified by extensible context-free grammars, we state and prove the validity of a recognition algorithm, and we outline how this may be converted to a parse algorithm. We conclude the chapter with a discussion of the extensible context-free formalism, some open problems in its theory, and its application to the broader issue of constructing extensible programming languages.

Study of the design and formal specification of a base language perhaps requires some justification. It might, for example, be argued that any language can be made extensible by grafting on extension facilities;

hence, one might (erroneously) conclude that the question of a base language is vacuous. It is true, of course, that many languages can be improved by such grafting. Further, in some circumstances desire for compatibility may dictate that a language in extensive use be so upgraded rather than replaced. However, where such considerations are not overriding, it seems decidedly advantageous to employ a base language designed with extensibility in mind. Conventional programming languages lack, often explicitly deny, the generality required of an extensible base. At the same time, such languages are frequently far more complex than a base language should be. One could, in principle, start with a conventional language, generalize by removing restrictions, simplify by removing special case mechanisms, and arrive at an acceptable base. It seemed easier, however, to start afresh. It was our belief that by so doing, we could improve considerably upon existing languages and produce a language more simple and parsimonious, yet more general and powerful. We have designed such a base language, named "EL1". Chapter 3 discusses its design and formal specification.

Concern with a formal specification perhaps also requires justification. In describing a programming language, the main goal is to explain how to write programs in it and what such programs mean. The former is syntactic specification, the latter, semantic specification. With relatively unimportant exceptions, syntax is satisfactorily specified in the framework of context-free grammars. These are suitable not only for human consumption but also as the basis for mechanical parsing. Further, as discussed in chapter 2, they can be cleanly augmented to handle languages which grow by the addition of new syntax rules. By and large, we have a good handle on syntactic specification. However, turning to semantic

specification, we find no parallel success. Most programming languages have their semantics specified by natural language descriptions (usually in English). These are generally imprecise, ambiguous, lacking in detail, and otherwise unsatisfactory. Even after careful reading of such a defining report, uncertainty as to the meaning of one or more constructions is generally the rule rather than the exception. On the other hand, repeated attempts[†] to define programming languages in terms of semantic formalisms such as the λ-calculus or Markov algorithms have fared no better. Hence, the formal semantic specification of programming languages is a nontrivial matter of considerable importance.

We wish to emphasize that this thesis embraces only two topics of a potentially far larger study of extensible languages. Several important issues still require solution, many others invite investigation, and doubtlessly still others will be uncovered as simpler problems are solved. Two that seem important at this point are (1) relating a simple facility for operator definition to the syntax mechanism, and (2) global resolution of meaning. We will outline these in turn.

While an extensible context-free grammar provides complete control over the syntax, it may prove more powerful than appropriate and hence somewhat awkward to use for simple cases. For example, a new arithmetic infix operator "op", with binding strength between "+" and "*", could be added to Algol 60 by redefining the necessary productions;[‡] however, it

---

[†]These various attempts are reviewed in section 2.1 of chapter 3.

[‡]Specifically, in section 3.3.1 of [Naur63], we delete

⟨term⟩ ::= ⟨factor⟩ | ⟨term⟩ ⟨multiplying operator⟩ ⟨factor⟩

and add

⟨term⟩ ::= ⟨term2⟩ | ⟨term⟩ op ⟨term2⟩

⟨term2⟩ ::= ⟨factor⟩ | ⟨term2⟩ ⟨multiplying operator⟩ ⟨factor⟩

would be preferable to simply define

operator op priority + < op < × means . . .

and have the appropriate changes to the syntax generated automatically. Allowing this facility and generalizing it to the various syntactic forms which may be regarded as "distributed" operators (e.g., Algol's if-then-else) requires some study. In general, the construction of face-plates which make complicated mechanisms easily available for simple uses is a matter which deserves attention.

Global resolution of meaning refers to a possible generalization of the Galler-Perlis paper discussed earlier. Given an expression composed of objects of some defined data type and operators acting on these objects, e.g.,

$$A_1 \ op_1 \ (A_2 \ op_2 \ A_3)$$

it is frequently the case that direct application of operators to operands is wasteful of some computer resource. We should like to allow the action of op_2 to be nonorthogonal to op_1. In general, we should like to allow the meaning of a form and its evaluation to depend upon the context in which it appears. Galler and Perlis treat the case where the interaction is a parse-tree rearrangement intended to optimize the amount of temporary storage required. As noted in a codicil to their paper, certain circumstances require the optimization of other resources (e.g., time), which may be carried out by different sets of transformation rules. They point out: "The important lesson here is that one should have available not only a variety of definitions, but a variety of substitution and tree-arrangement strategies." Investigation of such strategies and their generalization in a system which allows the programmer to specify transformation rules

requires considerable study. While this area is presently ill-defined and ill-developed, we believe it will eventually prove a source of significant power in extensible programming languages.

REFERENCES

[Back57] Backus, J.W. "The FORTRAN Automatic Coding System," Proceedings of the Western Joint Computer Conference, Vol. 11, 1957, pp. 188-198.

[Barr63] Barron, D.W. et al. "The Main Features of CPL," The Computer Journal, Vol. 6 (1963), pp. 134-143.

[Beech70] Beech, David. "A Structural View of PL/I," Computing Surveys, Vol. 2, No. 1 (March 1970), pp. 33-64.

[Chea66] Cheatham, T.E. "The Introduction of Definitional Facilities Into Higher Level Programming Languages," AFIPS Fall Joint Computer Conference, 1966, Vol. 29, pp. 623-637.

[Chom65] Chomsky, Noam. Aspects of the Theory of Syntax, The M.I.T. Press, Cambridge, Massachusetts, 1965.

[Chris65] Christensen, Carlos. "Examples of Symbol Manipulation in the AMBIT Programming Language," Proc. ACM 20th National Conference, Cleveland, Ohio, August 1965, pp. 247-261.

[Dahl66] Dahl, O. and Nygaard, K. "SIMULA – An ALGOL-Based Simulation Language," Comm. ACM, Vol. 9, No. 9 (September 1960), pp. 671-682.

[Gall67] Galler, B.A. and Perlis, A.J. "A Proposal for Definitions in ALGOL," Comm. ACM, Vol. 10, No. 4 (April 1967), pp. 204-219.

[Gar67] Garwick, Jan V. A General Purpose Language, Intern rapport S-32, Norwegian Defence Research Establishment, June 1967.

[Garb66] Garber, D. et al. "The SNOBOL3 Programming Language," Bell System Technical Journal, 45 (1966), pp. 895-943.

[Leav66] Leavenworth, B.M. "Syntax Macros and Extended Translation," Comm. ACM, Vol. 9, No. 11 (November 1966), pp. 790-793.

[Naur63] Naur, P. (ed.). "Revised Report on the Algorithmic Language ALGOL 60," Comm. ACM, Vol. 6, No. 1 (January 1963), pp. 1-17.

[Parn66] Parnas, David L. "A Language for Describing the Functions of Synchronous Systems," Comm. ACM, Vol. 9, No. 2 (February 1966), pp. 72-76.

[Perlis66] Perlis, Alan J. et al. A Definition of Formula Algol, Center for the Study of Information Processing, Carnegie Institute of Technology, 1966.

[Perlis66b] Perlis, A.J. "The Synthesis of Algorithmic Systems," Proc. of the 21st National Conference, Association for Computing Machinery (1966), Thompson Book Company, Washington, D.C., pp. 1-6.

[Rad65] Radin, George and Rogoway, H. Paul. "Highlights of a New Programming Language," Comm. ACM, Vol. 8, No. 1 (January 1965), pp. 9-17.

[Ruti67] Rutishauser, Heinz. Description of ALGOL 60, Springer-Verlag, New York, 1967.

[Smith60] Smith, Joseph W. "Syntactic and Semantic Augments to ALGOL," Comm. ACM, Vol. 3, No. 4 (April 1960), pp. 211-213.

[Stand69] Standish, Thomas A. "Some Features of PPL, A Polymorphic Programming Language," in Proc. of the Extensible Language Symposium, in SIGPLAN Notices, Vol. 4, No. 8 (August 1968), pp. 20-26.

[Strac62] Strachey, C. Speaking during a panel discussion: "Is a Unification ALGOL-COBOL ALGOL-FORTRAN Possible? The Question of One or Several Languages," in Symbolic Languages in Data Processing (Proceedings of the Symposium of the International Computation Center, Rome, 1962), Gordon and Breach Science Publishers, New York, 1962.

[vanD62] van der Poel, W.L. Speaking during a panel discussion: "Reflections from Processor Implementors on the Design of Languages," in Symbolic Languages in Data Processing, (Proceedings of the Symposium of the International Computation Center, Rome, March 1962), Gordon and Breach Science Publishers, New York, 1962.

[Witt22] Wittgenstein, Ludwig. Tractatus Logico-Philosophicus, Routlegge and Kegan Paul Ltd., London, 1922.

Chapter 2

EXTENSIBLE CONTEXT-FREE LANGUAGES

Section 1. INTRODUCTION

In this chapter we present a class of grammars designed for the syntactic description of extensible programming languages. These grammars employ a departure from conventional syntactic formalisms in that their syntax is not fixed, but rather is made variable.

This notion is best introduced by a review of the simpler case. In a conventional grammar, there is a fixed body of syntax rules and a set of instructions for using these rules to generate the legal strings of the language. Various types of syntax rules and various sets of instructions for using these rules form classes of grammars. The most familiar instance of this descriptive method is Algol [Naur63], whose grammar is of the class "context-free". Its syntax rules are all rewriting rules ($\alpha \rightarrow \beta$) with a single symbol on the left-hand side of the replacement arrow. The single, implicit instruction is a replacement rule: if $\alpha A \beta$ is an intermediate string and $A \rightarrow \gamma$ is a rule, then $\alpha \gamma \beta$ is an intermediate string. Many other classes of grammars have been proposed for the description of programming and natural languages. All share a common trait: for each grammar, the set of syntax rules is a fixed, finite set. This is satisfactory so long as the language has a fixed, predetermined syntax.

We assume the reader is familiar with the notion of an extensible programming language, e.g., [Bell68], [Chea66], [Chea68], [Gall67],

[Gar68]. By this term, we mean a higher level language which includes mechanisms with which the user can extend the language to facilitate its use in various application areas. One useful facility of such a language is a means whereby new syntactic forms can be added to the language for local use. To take a concrete – and somewhat restricted – example, suppose such a facility were added to Algol, resulting in a new language called Algol-E. In Algol, one may declare new variables and new procedures in the blockhead of each block. In Algol-E, one may additionally declare new syntax rules whose scope is that block and all blocks it contains. It might be useful to allow deletion of existing productions as well as addition of new ones, so that for each block the set of production rules, P_i, is given by:

$$P_i = P_{io} \cup P_{ia} - P_{id},$$

where P_{io} is the production set of the immediately containing block, P_{ia} is the set of productions declared in the blockhead as added, and P_{id} is the set of productions declared as deleted.

For example, a block with a restricted for statement might contain a declaration:

production ⟦⟨for list element⟩ → ⟨arithmetic expression⟩ to ⟨arithmetic expression⟩ ⟧,

⟦⟨for list element⟩ ↛ ⟨arithmetic expression⟩ step ⟨arithmetic expression⟩ until ⟨arithmetic expression⟩ ⟧.

Each production is enclosed in the brackets "⟦" and "⟧"; a right arrow indicates a rule is being added; a slashed arrow indicates a rule being deleted from the production set. Within the block, forms like

"for i := 1 to n do . . ." are legal, while the more general form "for i := 1 step s until n do . . ." is not.

For new forms to be useful, it is necessary to specify semantics as well as syntax. Associated with each production being added would be a definition of its meaning expressed in terms of the semantics of the enclosing block. This raises a set of issues concerning semantic variability. We will not deal with these issues in this chapter. Our interest here is in syntactic variability: how this can be formalized, what properties are thereby obtained, and how strings with variable syntax can be parsed.

While our primary interest is in the variability of syntactic forms per se, one special case is of particular interest. It will be recalled that the form of a legal Algol program is only partially specified by its context-free grammar; other restrictions are described by the English text. It has been shown that some of these restrictions cannot, even in principle, be expressed by a context-free grammar. In particular, the requirement that all variables be declared is such a restriction. It has been suggested, [DiFor63], that a declaration (e.g., real temperature;) may be regarded as specifying a new syntax rule (e.g., production [[⟨variable identifier⟩ → temperature]];). If this convention is adopted and the rule "⟨variable identifier⟩ → ⟨identifier⟩" is deleted from the syntax of Algol, then it is guaranteed that only declared variable names may be used in block bodies. It should be noted that such conventions cannot be used to specify all of the non-context-free restrictions. We have not, for example, made the necessary provision that a variable may not be declared in two conflicting ways. However, it is of interest

to note that a partial solution to a standing problem in language specification drops out as a special case of syntactic variability.

There is a substantial body of research by others (e.g., [Aho68], [Aho69a], [Aho69b], [Fisch68], [Gins67], [Gins68b], [Greib68], [Ros69], [Whit68a], [Whit68b], [Whit68c], [Whit69]) into classes of grammars which generalize the context-free. Much of this work has been motivated by a desire to model non-context-free restrictions on conventional programming languages. However, there has been little research into the formal properties of extensible languages. Bell [Bell68] describes an extensible language defined by a grammar belonging to a class he terms priority BNF. However, since priority BNF grammars generate the recursively enumerable sets,* most questions of interest, including membership, are undecidable.

*Although this is not discussed in Bell's work, it is entirely straightforward to show that any Turing machine can be imitated by a priority BNF grammar.

Section 2. THE FORMALISM

2.1 PRELIMINARY REMARKS

We now turn to a specific formalism which embodies the notion of variable syntax. We will define a class of grammars, the extensible context-free (ECF), which contains grammars such as those discussed in the previous section. Before doing so, however, we wish to generalize these examples somewhat.

The definition of local syntax in terms of a local production set for each block is clearly dependent upon Algol's block structure. In other languages, the block structure might be substantially modified, or entirely absent. To obtain a formalism which does not depend on language idiosyncrasy, we adopt the convention that string "structure" will be ignored, and new productions may be used anywhere in the string to the right of the point at which they are declared. This is made well-defined if rewriting rules are applied only to the leftmost nonterminal. (This is, of course, no restriction on the weak generative power of context-free grammars.) Hence, at any stage of a derivation the string will have the form $xA\alpha$, where x is a terminal string, A is a nonterminal, and α is an arbitrary string of terminals and nonterminals. The local production set is determined by the productions initially in the syntax and by those which appear in the string x. The local production set, in turn, specifies in what ways A may be rewritten.

The second generalization we shall make is that the new productions need not appear explicitly in the terminal string, so long as

they can be derived from it. Part of each ECF grammar will be a finite state machine with output, or finite state transducer. The finite state transducer (FST) takes the terminal string as input and outputs the associated set of productions. This mapping serves several functions, the most important of which is to allow the specification of productions involving nonterminal symbols by means of a string of terminals. For, by definition, a nonterminal cannot appear in a terminal string; yet each production must contain at least one nonterminal. The mapping also permits the formalism to cover extensible languages in which syntactic extensions are not stated explicitly as productions, but rather by means of macro forms or operator definitions. The latter forms may be distinctly preferable to some classes of users.

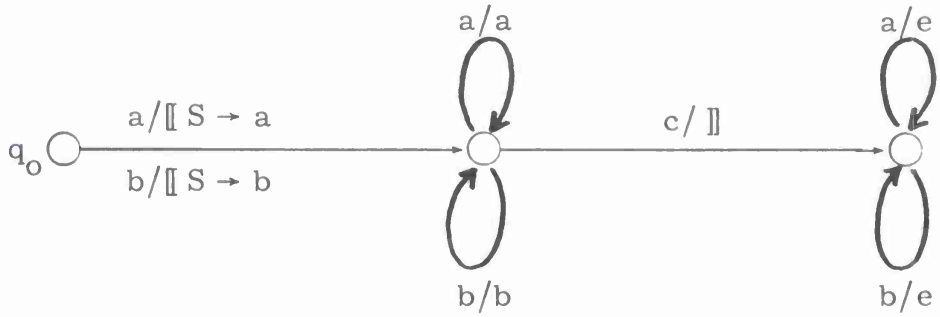
To coordinate the two activities – generation via the syntax rules and change of syntax rules due to the output of the FST – we amend the above description as follows. At any stage of derivation, let the string be $xA\alpha$. The local production set is given by: (1) the initial productions, and (2) the output of the FST given x as input.

An example may help make this clear. Consider a grammar with terminal vocabulary $\{a, b, c\}$, nonterminal vocabulary $\{X, F, S\}$, and initial production set:

$$X \rightarrow FcS$$

$$F \rightarrow aF \mid bF \mid a \mid b$$

The finite state transducer is specified by its state transition diagram:*



where q_o is the start state and all transitions not explicitly specified lead to a "dead" state. A sample derivation by this grammar is:

$$X \Rightarrow FcS \Rightarrow aFcS \Rightarrow abFcs \overset{*}{\Rightarrow} abaaabcS$$

At this point, the FST acting on the initial terminal string has output a complete production: $[\![S \rightarrow abaaab]\!]$. Hence, the local production set is:

$$X \rightarrow FcS$$

$$F \rightarrow aF \mid bF \mid a \mid b$$

$$S \rightarrow abaaab$$

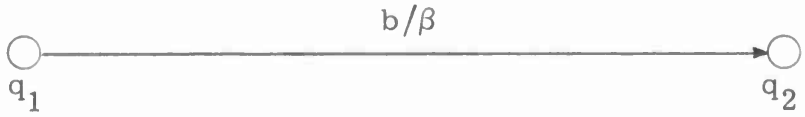
The derivation concludes with a final step

$$abaaabcS \Rightarrow abaaabcabaaab$$

Clearly, the language generated by the grammar is $\{wcw \mid w \in \{a,b\}^+\}$, which is known not to be context-free.

---

*A state transition diagram is interpreted as follows. The nodes represent states; the arcs between them represent transitions. Consider, for example,



This is read: when in state q_1 if the input is "b" then output "β" and go into state q_2.

One final generalization is required. In the above example, the nonterminal vocabulary was fixed, and the single new production employed one of these nonterminals on its left-hand side. A characteristic property of grammars is that they use a finite vocabulary, and in particular a finite number of nonterminals. For any language having a fixed syntax, this is quite acceptable: The set of productions being finite, the number of nonterminals is a fortiori also finite. However, if the production set is allowed to grow, any given finite set of nonterminals may be found to be too small. We wish to consider languages whose local syntax may be of arbitrary complexity; this, in general, requires an unbounded set of nonterminals. (We show, in Appendix I, that given a terminal vocabulary of more than two symbols and any integer k, there exists a context-free language such that any context-free grammar which generates the language uses at least k nonterminal symbols.)

To provide for an unbounded set of nonterminals and still work within a finite vocabulary, it is necessary to use some sort of encoding. This is, of course, precisely what is done in the Algol report. We may regard a syntax rule "⟨ letter ⟩ ::= a" either as a context-free production with right-hand side "a" and left-hand side the single nonterminal "⟨ letter ⟩", or with equal validity as a Type-0 production whose left-hand side consists of the eight symbols: "⟨ ", "l", " e", "t", "t", "e", "r", " ⟩ ". We shall take the latter viewpoint.

For any given ECF grammar, let the terminal vocabulary be Σ, let the total vocabulary be V, and let the FST have output vocabulary Δ, with $\Sigma \subseteq \Delta$. We assume that Δ includes a set of six distinguished symbols $\Gamma = \{ [\![,]\!], \rightarrow, \not\rightarrow, \langle , \rangle \}$ and that Γ is disjoint from Σ. Let $V_N = V - \Sigma - \Gamma$,

let $V_M = \{\langle w \rangle \mid w \in \Sigma^+\}$, and let $I = V_N \cup V_M$. New productions are taken to be those substrings of the FST output having the form

$$[\![A \rho \alpha]\!] \quad \text{where} \quad A \in I, \quad \rho \in \{\rightarrow, \not\rightarrow\}, \text{ and } \alpha \in (I \cup \Sigma)^*$$

We thereby place strict requirements on the form of substrings which may be rewritten. This guarantees that generation behavior is essentially context-free in the sense that information may not be passed along in the string. In accord with common parlance, we refer to I as the set of intermediate symbols, with the understanding that a member of I may actually be a character string.

2.2 FORMAL DEFINITION

In the preceding discussion, we made use of a number of notions in an informal fashion, depending on the reader's intuition for their meaning. We now proceed to give precise definitions of these notions. Our goal will be a formal definition of ECF languages.

Definition 2.2.1. A finite state transducer with accepting states (FST) is a 7-tuple $T = (K, \Sigma, \Delta, \delta, \lambda, q_o, F)$, where K is a finite set of states, $F \subseteq K$, where Σ and Δ are finite input and output vocabularies, respectively, and where $q_o \in K$ is the initial state. δ is the transition function δ: $K \times \Sigma \rightarrow K$, and λ is the output function λ: $K \times \Sigma \rightarrow \Delta^*$. The functions δ and λ are extended so that $\delta : K \times \Sigma^* \rightarrow K$ and $\lambda : K \times \Sigma^* \rightarrow \Delta^*$ by the following definitions:*

$$\delta(q, e) = \lambda(q, e) = e$$
$$\delta(q, xa) = \delta(\delta(q, x), a)$$
$$\lambda(q, xa) = \lambda(q, x)\, \lambda(\delta(q, x), a) \quad \forall\ x \in \Sigma^*, \quad a \in \Sigma, \quad q \in K.$$

*We use the symbol "e" to denote the empty string.

If $w \in \Sigma^*$, then T(w) is defined to be $\lambda(q_o, w)$; if $L \subset \Sigma^*$, then T(L) is defined to be $\{T(w) \mid w \in L\}$. If $w \in \Sigma^*$, we say T <u>accepts</u> w if $\delta(q_o, w) \in F$.

<u>Definition 2.2.2</u>. An extensible context-free (ECF) grammar is an 11-tuple $G = (V, \Sigma, P_o, X, T, \langle , \rangle , [\![,]\!], \rightarrow, \not\rightarrow)$, where V is a finite set of symbols, $\Sigma \subset V$ is the terminal vocabulary, $X \in V - \Sigma$ is the initial symbol, and $\Gamma = \{\langle , \rangle , [\![,]\!] , \rightarrow, \not\rightarrow\} \subseteq V - \Sigma$. We define $V_N = V - \Sigma - \Gamma$, $V_M = \{\langle w \rangle \mid w \in \Sigma^+\}$, $I = V_N \cup V_M$, and $\underline{V} = I \cup \Sigma$. P_o is a set of initial productions, each of the form $A \rightarrow \alpha$, where $A \in I$ and $\alpha \in \underline{V}^*$. Finally, T is a finite state transducer $T = (K, \Sigma , \Sigma' , \delta, \lambda, q_o, F)$ where $\Sigma' \subset V$.

<u>Remark</u>. For brevity of notation, the special symbols "$\langle$", "$\rangle$", "$[\![$", "$]\!]$", "$\rightarrow$", and "$\not\rightarrow$" will henceforth be assumed to be present, and an ECF grammar will be specified as a 5-tuple $G = (V, \Sigma, P_o, X, T)$.

Note that while V is the vocabulary, $\underline{V}$ is the effective vocabulary, for symbol strings of the form $\langle w \rangle$ (where $w \in \Sigma^+$) act as single elements.

Let G be an ECF grammar. The language, L(G), generated by G is defined by specifying (1) the form of an instantaneous description, and (2) the transitions which take an instantaneous description into its possible successors.

<u>Definition 2.2.3</u>. An instantaneous description (ID) of an ECF grammar $G = (V, \Sigma, P_o, X, T)$ is an element of $\Sigma^* \times \underline{V}^*$.

Let $\pi = (w, \gamma)$ be an instantaneous description. T(w) is the output of the finite state transfucer T for the ID π. This output, taken together with the initial production set P_o, determines the set, P_π, of legal productions applicable to π. We refer to P_π (or to P, when π is understood) as the <u>local production set</u>. Denoting the projection function

which maps π into its first component by U, we write:

$$P_\pi = \mathbb{P}(P_o, T(U(\pi))),$$

where $\mathbb{P}$ is specified by the following procedure.

The string $T(U(\pi))$ contains a unique set of disjoint substrings, each of the form:

$$[\![A \rho \alpha]\!] \qquad \text{where} \qquad A \in I,$$
$$\rho \in \{\rightarrow, \not\rightarrow\} \quad \text{and}$$
$$\alpha \in \underline{V}^*.$$

It is possible that $T(U(\pi))$ contains no such substrings. However, if it contains any, they are guaranteed to be uniquely defined and disjoint since $[\![,]\!] \notin \underline{V} \cup \{\rightarrow, \not\rightarrow\}$. There will be finitely many such substrings, say N; let them be indexed and let $\phi_i = A_i \rho_i \alpha_i$ for $i = 1, \ldots, N$. Then $P_1, P_2, \ldots P_N$ are defined as follows:

for $i = 1, \ldots N$

if $\rho_i =$ "$\rightarrow$" then $P_i = P_{i-1} \cup \{\phi_i\}$, else $P_i = P_{i-1} - \{\phi_i\}$.

Finally, define $P_\pi = P_N$.

The transition between an ID π and a successor π' is denoted by $\pi \Rightarrow \pi'$ and is obtained as follows:

(1) If $(A \rightarrow \alpha) \in P_\pi$, then

$$\pi = (w, A\beta) \Rightarrow (w, \alpha\beta) = \pi'.$$

(2) If $a \in \Sigma$, then

$$\pi = (w, a\beta) \Rightarrow (wa, \beta) = \pi'.$$

The extension of $\Rightarrow$ is denoted $\overset{m}{\Longrightarrow}$ and is defined by:

$\pi \overset{m}{\Longrightarrow} \pi'$ $(m \geq 0)$ if $\exists$ ID's $\pi_o, \pi_1, \ldots \pi_m$ such that

$\pi = \pi_o \Rightarrow \pi_1 \Rightarrow \ldots \Rightarrow \pi_m = \pi'$.

Finally, the transitive extension is denoted by $\overset{*}{\Longrightarrow}$ and is defined:

$$\pi \overset{*}{\Longrightarrow} \pi' \text{ if } \exists\ m\ (0 \leq m < \infty) \text{ such that } \pi \overset{m}{\Longrightarrow} \pi'.$$

Definition 2.2.4 Let $w = a_1 \ldots a_n \in \Sigma^*$. A derivation, Π, of w is a sequence of ID's $\Pi = \pi_o, \pi, \ldots \pi_m$ such that

(1) $\pi_o = (e, X)$,

(2) $\pi_i \Rightarrow \pi_{i+1}$ $\quad i = 0, \ldots m-1$,

(3) $\pi_m = (w, \gamma)$ for some $\gamma \in \underline{V}^*$.

A derivation $\Pi = \pi_o \ldots \pi_m$ is said to be terminal if $\pi_m = (w, e)$.
A derivation $\Pi = \pi_o \ldots \pi_m$ is said to use a production $A \rightarrow \alpha$ if
$\exists\ i\ (0 \leq i \leq m-1)$ such that $\pi_i = (w, A\gamma)$ and $\pi_{i+1} = (w, \alpha\gamma)$ for some γ.

Remark. When speaking informally, it will frequently be useful to write an ID $\pi = (w, \beta)$ in the simpler form "$w\beta$". Analogously, a derivation $(e, X) \overset{*}{\Longrightarrow} (w, \gamma)$ will sometimes be written "$X \overset{*}{\Longrightarrow} w\gamma$". Context will make clear which use of the transition symbol is intended.

Definition 2.2.5. Let $G = (V, \Sigma, P_o, X, T)$ be an ECF grammar. The language generated by G is defined to be:

$$L(G) = \left\{ w \mid (e, X) \overset{*}{\Longrightarrow} (w, e) \text{ and } \delta_T(q_o, w) \in F_T \right\},$$

where δ_T and F_T are the transition function and accepting states of T.

The above definition of instantaneous description and ID transition has the virtue of simplicity, but viewed as a computational procedure it is incredibly inefficient. It blithely ignores an essential property of ECF derivations: i.e., if $\Pi = \pi_o \ldots \pi_m$ is a derivation, then $U(\pi_i)$ is monotone nondecreasing as a function of i. This monotonicity makes it possible to compute P_{π_i} by incremental techniques, adjusting the local

production set as new productions are added to the right end of $T(U(\pi_i))$. In Section 4.2, we will discuss an alternate development of instantaneous descriptions which makes use of this property.

2.3 EXAMPLES

A few examples may help to illustrate the generative power of the above formalism. The first of these will be frequently used in later discussion.

In these examples and elsewhere in this paper, it will be convenient to specify an FST by means of a state transition diagram instead of by an explicit definition of its transition and output functions. These diagrams will be simplified if we adopt the convention that all unspecified transitions lead to a dead state. The dead state emits no output (i.e., gives the empty string as output) and is a nonaccepting state (i.e., does not belong to F_T). Also, unless specifically stated otherwise, all states explicitly shown in a state transition diagram are accepting states.

Example 2.3.1 (non-primes ≥ 4 preceded by a factor).

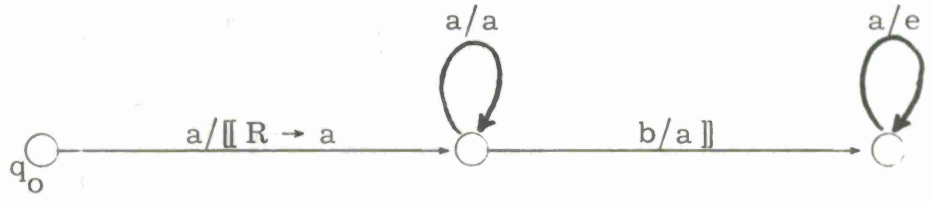
The language $\{a^n b a^{(n+1)m} \mid n \geq 1,\ m \geq 2\}$ is generated by $G = (V, \Sigma, P_o, X, T)$, where $V = \{X, N, A, R, a, b, [\![,]\!], \rightarrow, \not\rightarrow, \langle, \rangle\}$, $\Sigma = \{a, b\}$, and P_o is given by:

$$X \rightarrow AbRN$$

$$A \rightarrow aA \mid a$$

$$N \rightarrow RN \mid R .$$

The finite state transducer is specified by:



This grammar, which is similar to Fischer's Example 1.2.2 [Fisch68], generates a string of one or more a's, followed by two or more repetitions of the initial string of a's.

Example 2.3.2 (a very simple algebraic language in which variables must be declared).

The initial productions are:

⟨block⟩ → ⟨blockhead⟩ ; ⟨compound tail⟩

⟨blockhead⟩ → begin ⟨declaration⟩ | ⟨blockhead⟩ ; ⟨declaration⟩

⟨declaration⟩ → declare ⟨name⟩

⟨name⟩ → ⟨letter⟩ | ⟨letter⟩ ⟨name⟩

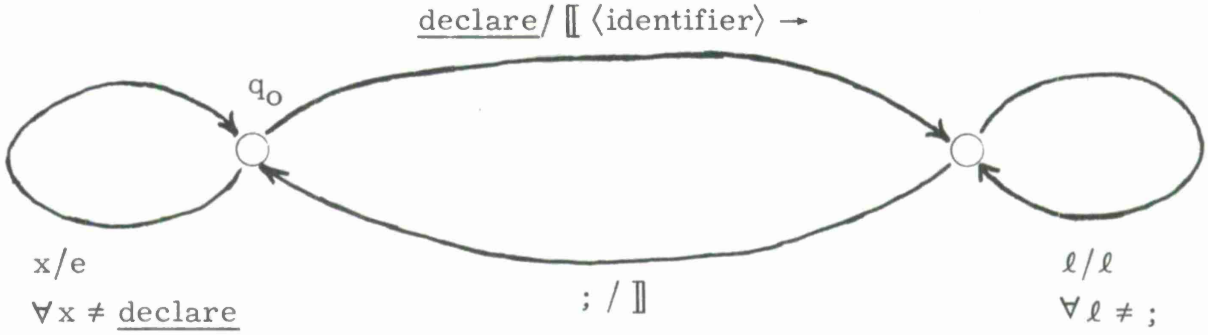
⟨letter⟩ → a | b | c | . . . | y | z

⟨compound tail⟩ → ⟨statement⟩ end | ⟨statement⟩ ; ⟨compound tail⟩

⟨statement⟩ → ⟨identifier⟩ := ⟨expression⟩

⟨expression⟩ → ⟨identifier⟩ | ⟨identifier⟩ + ⟨expression⟩ .

It will be noted that there are no rewriting rules with "⟨identifier⟩" as left-hand side. This, however, is remedied by the FST:



This grammar is a particularly simple form of the Algol-like grammar discussed in the Introduction. The form ⟨declarations⟩ consists of the symbol "declare", followed by a string over the alphabet $\{a, b, c, \ldots z\}$, delimited by a semicolon. For each such declaration, a new production is adjoined to P.

Example 2.3.3 (the encodement of a context-free grammar followed by a string generated by that grammar).

This example is a schema for a set of ECF grammars, one for each possible terminal vocabulary. Let $\overline{\Sigma}$ be a (finite) terminal vocabulary. We construct an ECF grammar which generates strings consisting of the encodement of an arbitrary context-free grammar with terminal vocabulary $\overline{\Sigma}$, followed by a string belonging to that context-free language.

Let $G = (V, \Sigma, X, P_o, T)$ where $\Sigma = \overline{\Sigma} \cup \{\not<, \not>, \$\}$, and where $V = \Sigma \cup \{X, E, R, N, M, L, S, A, <, >, \rightarrow, \not\rightarrow, [\![,]\!]\}$. P_o is given by:

$X \rightarrow E < \sigma >$ for some $\sigma \in \overline{\Sigma}^+$

$E \rightarrow R\,E \mid R$

$R \rightarrow N\,S\,\$$

$N \rightarrow \not< M \not>$

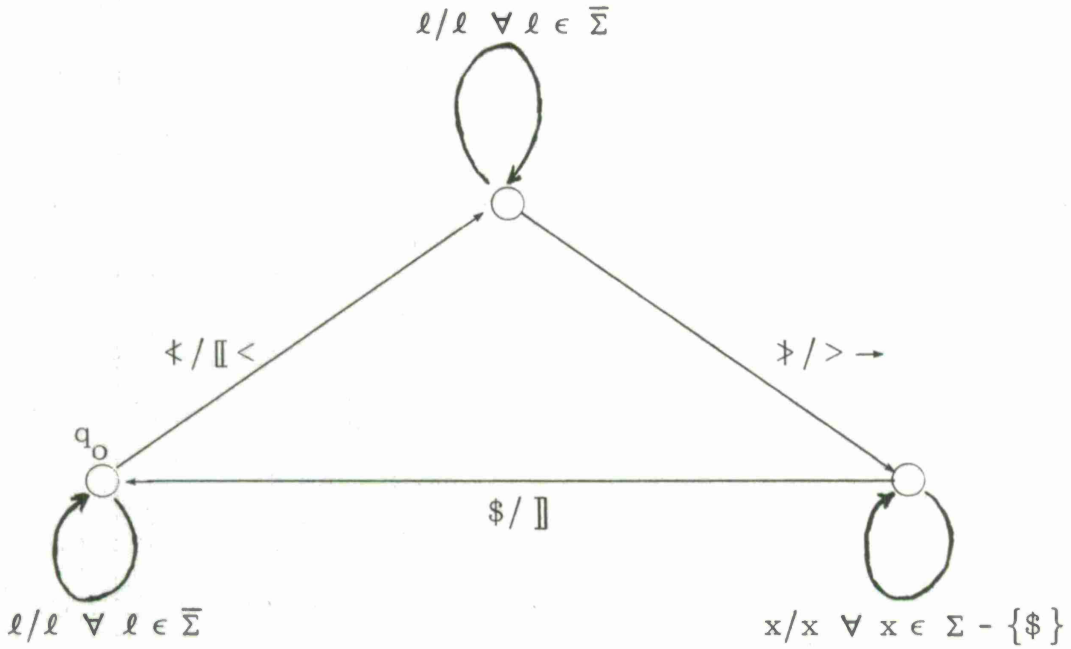
$M \rightarrow LM \mid L$

$L \rightarrow \ell \qquad \forall\, \ell \in \overline{\Sigma}$

$S \rightarrow AS \mid e$

$A \rightarrow N \mid L$

The finite state transducer is specified by the following diagram:



The grammar operates as follows. It first generates a sequence of substrings, each of the form

$$¢\, w \,\$\, \alpha \,\$ \qquad \text{where } w \in \overline{\Sigma}^{+},$$

$$\text{and } \alpha \in (\overline{\Sigma} \cup ¢\, \overline{\Sigma}^{+} \,\$)^{*},$$

each interpreted as a production

$$< w > \rightarrow \alpha .$$

This results in a local production set $P = P_o \cup P'$. Then a string is generated by a context-free derivation from the production set P'. For any context-free grammar G_c having terminal vocabulary $\overline{\Sigma}$, and

any $w \in L(G_c)$, there is some string in this language consisting of an encodement of G_c followed by w.

Remark. By suitable modification of this ECF grammar, it is possible to restrict the set of productions P′ to any of the following classes: regular productions, linear productions, intermediate constituent form, or Greibach normal form.

Section 3. FORMAL PROPERTIES

In this section, the formal properties of ECF languages will be explored. We study some of the usual characteristics of formal languages: closure under various operations, the membership problem, the emptiness problem, and a few related questions. We then examine a number of possible restrictions on the form of new productions and show that these lead to restricted classes of languages, thereby giving negative answers to certain questions concerning canonical form. Finally, we relate ECF grammars to other generalizations of the context-free.

3.1 STRUCTURAL PROPERTIES

We begin with a characterization of those productions output by the FST which make the associated language ECF but not CF (i.e., inherently ECF).

Definition 3.1.1. Let $G = (V, \Sigma, X, P_o, T)$ be an ECF grammar. If $w \in \underline{V}^*$, the length of w, $|w|$, is defined as follows:

(1) $|e| = 0$

(2) $|a| = 1 \quad \forall\, a \in \Sigma \cup V_N$

(3) if $A \in V_M$, $A = \langle a_1 \ldots a_n \rangle$ where $a_i \in \Sigma$, then $|A| = n + 2$

(4) if $w = w_1 w_2$ where $w_1, w_2 \in \underline{V}^*$, then $|w| = |w_1| + |w_2|$.

Definition 3.1.2. Let $\phi = A \rightarrow \alpha$ be a production. The length of ϕ, $|\phi|$, is defined to be $|\phi| = |A| + |\alpha|$.

We next define the term "rule length bounded" as applied to a variety of objects, culminating in the definition of a rule length bounded grammar.

Definition 3.1.3.

(1) Let $w \in L(G)$, and let Π be a terminal derivation of w. Π is rule length bounded with constant k (RLB-k) if, for each production ϕ used in Π, $|\phi| \leq k$.

(2) Let $w \in L(G)$. The string w is RLB-k if $\exists$ a terminal derivation, Π, of w which is RLB-k.

(3) Let G be an ECF grammar. G is RLB-k if $\forall\ w \in L(G)$, w is RLB-k. G is rule length bounded if it is RLB-k for some integer k.

Our first theorem asserts that if a grammar is rule length bounded, then its language is only context-free. Loosely speaking, this shows that a context-free grammar given the additional power to add new productions which are RLB is still only context-free.

Theorem 3.1.1. If G is a rule length bounded ECF grammar, then L(G) is a context-free language.

Proof (by pda argument).

Let $G = (V, \Sigma, X, P_o, T)$ be RLB-k. Let the number of elements in V, #(V), be N. Then the number of possible distinct productions is bounded by N^k. Hence, the number of possible distinct production sets is bounded by 2^{N^k}. Since this is finite, we can construct a

nondeterministic pushdown automaton* which accepts precisely the language L(G).

We sketch the construction. The action of the pushdown automata (pda) is nondeterministic, top-down, predictive. Corresponding to a production $A \rightarrow \alpha$, we have the pda step $(q, w, yA) \vdash (q, w, y\alpha^R)$ for an appropriate state q. We need only show that it is possible to determine the appropriate states; i.e., that the finite state control can keep track of which productions are valid at any point in the derivation.

The finite state control is essentially a cross-product construction of N^k components, in which the "active" productions are recorded one production per component. The start state corresponds to the production set P_0. As each symbol of input is read, the FST mapping is imitated. The FST output, which represents a sequence of productions, is reflected in the machine state. For each production added or deleted, a record is made in the corresponding cross-product component.

So constructed, the pda performs at random some legal generation (legal in the ECF sense) of the grammar G. The pda accepts if and only if the string so generated is identical to the input string. □

Remark. The converse does not hold. There exist ECF languages which are context-free but which are defined by ECF grammars that

---

*In the course of this paper, we will use a number of standard types of automata. Since these automata frequently appear in the literature, we assume familiarity with them on the part of the reader. However, since there is no universally accepted notation for these machines, this is a source of possible confusion. Hence, in Appendix II, we give the definitions and notations used in this paper for these automata.

are not rule length bounded. For example, consider $\{a^n c a^n | n \geq 1\}$, defined by a grammar which uses the first string of a's to form a production used to generate the second string of a's.

Suppose some ECF grammar is not rule length bounded. It will, however, contain some subset which is. This subset is context-free.

Definition 3.1.4. Let G be an ECF grammar and let k be an integer. We define L(G)/k as

$$\{w \in L(G) \mid w \text{ is RLB-k}\}.$$

Corollary 3.1.2

For any ECF grammar G, and any integer k, L(G)/k is context-free.

Corollary 3.1.3

Let G be an ECF grammar whose language, L, is not context-free. Then $\forall$ k, L - L/k is infinite.

Proof

Suppose the contrary. If for some k, L - L/k were finite, then there exists a finite set of ad hoc rules which produce L - L/k. Adjoin this set of rules to a context-free grammar which generates L/k. This yields a context-free grammar for L. Contradiction. □

It is well known that any context-free grammar whose terminal vocabulary is a single letter generates a regular set. Using this result and the above theorem, it can be shown that an identical result holds for ECF grammars. The idea is straightforward: if the terminal vocabulary is a single letter, there is a bound on the length of productions emitted by the FST. Hence, the grammar must be rule length bounded.

Since the language is therefore context-free, it is also regular. We need only show that the productions are, in fact, bounded in length.

Theorem 3.1.4

Let $G = (V, \{a\}, P_o, X, T)$ be an ECF grammar, where $T = (K, \{a\}, \Delta, \delta, \lambda, q_o, F)$. Then there exists an integer k such that if $\eta_1 [\![A \rightarrow \beta]\!] \eta_2 \in T(L(G))$, then $|A \rightarrow \beta| \leq k$.

Proof

We show the stronger result: if $\eta_1 [\![A \rightarrow \beta]\!] \eta_2 \in T(a^*)$, then $|A \rightarrow \beta| \leq k$. Since $L(G) \subseteq a^*$, the desired result follows from this.

Let $q^i = \delta(q_o, a^i)$. Since T is deterministic, this is well defined for all i. Consider the infinite sequence $q^1, q^2, \ldots q^n, \ldots$. Let the number of elements in K be denoted by #(K). For some i ($i \leq (\#K)$) and some j ($j \leq \#(K)+1$), we have $q^i = q^j$ and $i < j$. Let $p = j - i$. Since T is deterministic, it must repeat the cycle; hence,

$$q^{i+k} = q^{i+tp+k} \qquad \forall\ t, k \geq 0.$$

The infinite sequence of states must have the form

$$q^1 q^2 \ldots q^{i-1} (q^i q^{i+1} \ldots q^{i+p-1})^* .$$

Any finite sequence whose length exceeds i must have the form

$$q^1 q^2 \ldots q^{i-1} (q^i q^{i+1} \ldots q^{i+p-1})^n q^i q^{i+1} \ldots q^{i+s}$$

with $n \geq 0$ and $0 \leq s \leq p-2$. Since the output depends only on the state, let $\rho^\ell = \lambda(q^\ell, a)\ \forall\ \ell$. Then any output string produced by T whose length exceeds $|\rho^1 \ldots \rho^i|$ must have the form

$$\rho^1 \ldots \rho^{i-1} (\rho^i \ldots \rho^{i+p-1})^n \rho^i \ldots \rho^{i+s}$$

for some $n \geq 0$, $0 \leq s \leq p-2$.

If such a string contains a substring $m_1 \delta m_2$ where $m_1, m_2 \in V$, $\delta \in V^*$ and m_1 and m_2 are not in δ, then $|\delta| < |\rho^1 \ldots \rho^{i+p-1}|$.

Letting $m_1 = [\![$, $\delta = A \rightarrow \beta$ and $m_2 =]\!]$, this gives the result claimed. □

Corollary 3.1.5

Let G be an ECF grammar whose terminal vocabulary is a single symbol, then L(G) is regular.

3.2 CLOSURE

In this section we will examine the closure behavior of the family of ECF languages, under various operations. We show closure under several standard operations and under an operation which may be interpreted as reversible translation. However, we also show non-closure under homomorphism (even non-erasing). Hence, the family of ECF languages does not form an AFL (i.e., abstract family of languages, cf. [Gins68a]).

Theorem 3.2.1

Let G be an ECF grammar, let $L = L(G)$, and let R be a regular set. Then the following are ECF languages:

(a) $L \cap R$

(b) $L \cup R$.

Proof

(a) $L \cap R$ is a standard cross-product construction. Its grammar is that of L with one modification: the states of the new FST have an additional component which imitates the action of a regular

automaton that accepts R. The modified FST accepts if and only if both the regular automaton and the old FST would have accepted.

(b) $L \cup R$ is a variant of the above construction. Its grammar is obtained by making two modifications to G. (1) To P_o is added a set of productions, P_o', which generate Σ^*. If these productions are written using symbols not in the output vocabulary of T, it is guaranteed that members of P_o' will never be deleted and that there will be no interaction between these and other rules. (2) The states of the new FST have an additional component which is used to imitate the action of a regular automaton that accepts R. The new FST accepts if either the regular automaton or the old FST would have accepted.

Theorem 3.2.2

The family of ECF languages is not closed under homomorphism (even length-preserving homomorphism).

Proof

Example 2.3.1 demonstrates that $L = \{a^n b a^{(n+1)m} \mid n \geq 1,\ m \geq 2\}$ is an ECF language. Let $h: \{a, b\} \rightarrow \{a\}$ be a homomorphism, defined by $h(a) = h(b) = a$. Let $L' = h(L) = \{a^{pq} \mid p \geq 2,\ q \geq 3\}$. This consists of all possible strings of a's whose length is non-prime and greater than or equal to six. Clearly, L' is not regular. Hence, by Corollary 3.1.5, L' is not ECF. □

Corollary 3.2.3

The family of ECF languages is not an AFL [Gins68a].

Definition 3.2.1. Let L_1 and L_2 be languages. The left quotient of L_2 by L_1 is defined to be

$$L_1 \backslash L_2 = \{y \mid \exists\ x \in L_1 \text{ such that } xy \in L_2\}.$$

Corollary 3.2.4

The family of ECF languages is not closed under left quotient by regular sets.

Proof

Let $L = \{a^n b a^{(n+1)m} \mid n \geq 1,\ m \geq 2\}$, and let $R = a^*b$. Then $R \backslash L = \{a^{rs} \mid r, s \geq 2\}$ is not regular and thus by Corollary 3.1.5 is not ECF. □

Given that the family of ECF languages is not closed under homomorphism, even length-preserving, is is natural to ask if there is any class of mappings which insures closure. We observe that non-closure under the homomorphism $h(a) = h(b) = a$ is due to the identification of two formerly distinct symbols (i.e., due to loss of information). Hence, we conjecture that if a mapping preserves information, it will preserve the ECF property. Under suitable definition of "information preservation", this is indeed the case.

Definition 3.2.2. A homomorphism $h: \Sigma \to \Delta$ is said to be invertible if $\exists$ a generalized sequential machine* (GSM), g, such that $\forall\ w \in \Sigma^*$, $g(h(w)) = w$.

---

*A GSM is a finite state transducer in which all states are accepting states.

Lemma 3.2.5

If h is an invertible homomorphism, then it is non-erasing.

Proof

Suppose the contrary: i.e., $h(a) = e$ for some $a \in \Sigma$, $a \neq e$. Then $\forall x \in \Sigma^*$, $h(x) = h(xa)$. In particular, for $x = e$, we have $h(e) = h(ea)$. Letting g be an inverting GSM for h, $e = g(h(e)) = g(h(ea)) = ea = a$. So $e = a$, contrary to assumption. □

Theorem 3.2.6

Let $G = (V, \Sigma, P_o, X, T)$ be an ECF grammar and let $h: \Sigma \rightarrow \overline{\Sigma}^*$ be an invertible homomorphism. Then $h(L(G))$ is an ECF language.

Proof

A grammar $\overline{G} = (\overline{V}, \overline{\Sigma}, X, \overline{P}_o, \overline{T})$ which generates $h(L(G))$ is obtained as follows. Let $\overline{V} = V \cup \overline{\Sigma}$. Extend h so that $h: V \rightarrow \overline{V}^*$ as follows: if $b \in \Sigma$ then $h(b)$ is already defined; if $s \notin \Sigma$ then $h(s) = s$. $\overline{P}_o$ is obtained from P_o by applying h to each production. For example, if $\Sigma = \{a, b\}$, $V_N = \{A, B\}$, and $P_o = \{A \rightarrow a, \langle a \rangle \rightarrow abB\}$, then $\overline{P}_o = \{A \rightarrow h(a), \langle h(a) \rangle \rightarrow h(a)h(b)B\}$.

Let g be a GSM which inverts h. $\overline{T}$ is defined to be $h \circ T \circ g$, under the operation of functional composition. ($\overline{T}$ accepts if and only if the image of T which it contains accepts.)

For each instance of a terminal symbol, b, in a derivation of G, a corresponding instance of h(b) will appear in a derivation of $\overline{G}$. $\overline{T}$ inverts h(b) to recover b, imitates the action of T on b, and applies h to the output generated. Hence, $h(w) \in L(\overline{G})$ if and only if $w \in L(G)$. □

3.3 RELATION TO THE FAMILY OF CONTEXT-SENSITIVE LANGUAGES

The membership problem is said to be solvable for a family of languages if there exists a procedure which, given any language L of the family and any string w, decides whether or not $w \in L$. In Sections 4.3 and 4.5, we will present and prove the validity of a recognition algorithm for the ECF languages, thus showing that their membership problem is solvable.

Having shown that $w \stackrel{?}{\in} L(G)$ can be decided by a Turing machine, we next ask whether this can still be done by a Turing machine in space n; i.e., on a linear bounded automata (lba). The same question, stated in terms of languages, is whether ECF languages are context-sensitive. For the general case, the question is open. As will be shown, we can demonstrate a procedure which works in space n^2 but not in n. However, for a large class of ECF grammars, we can exhibit containment in the context-sensitive (CS). We will define this class, prove the assertion, and then discuss space n^2.

Definition 3.3.1. A production $A \rightarrow \alpha$ is said to be L-restricted if $|A| \leq |\alpha|$. A derivation is L-restricted if all productions used in the derivation are L-restricted. An ECF grammar G is L-restricted if $\forall\ w \in L(G)\ \exists$ an L-restricted terminal derivation of w.

Remark. A production $A \rightarrow \alpha$ is clearly L-restricted if $A \in V_N$ and $\alpha \neq e$. The significance of the L-restriction is that it guarantees that a derivation does not involve a "swell" of substrings belonging to V_M. That is, under the L-restriction, if $(w, \beta) \stackrel{*}{\Longrightarrow} (ww', e)$, then $|\beta| \leq |w'|$.

Theorem 3.3.1

Let $G = (V, \Sigma, X, P_o, T)$ be an L-restricted ECF grammar, then L(G) is a context-sensitive language.

Proof

We construct a nondeterministic lba, M, which performs a legal derivation of G and accepts if and only if the string so generated is identical to its input. From this, it follows directly that L(G) is context-sensitive.

M's tape is divided into three tracks: T1, T2 and T3. T1 contains the input while T2 and T3 are working tracks. Letting "$\not b$" be a new symbol reserved to designate a blank tape square, the initial configuration is:

$$¢\left\{\begin{array}{c} w \\ \not b^{|w|} \\ \not b^{|w|} \end{array}\right\} \$$$

M begins its operation by imitating the action of T acting on w, writing the output corresponding to each symbol of T1 directly beneath it on T3. This requires (1) a symbol reserved to indicate e-output, (2) possible compression by a factor of k, where k is the length of the longest string emitted by T for any single input symbol. Since k is fixed for the grammar, this compression presents no problem and is subsequently ignored. As M imitates T, it keeps track of T's state. When all of w has been processed, M rejects if the simulated state of T is not an accepting state. After this first step has been completed, M initializes T2 to the start symbol X, so that the tape contains

$$¢ \left\{ \begin{matrix} w \\ X \\ T(w) \end{matrix} \right. \qquad \left. \begin{matrix} \\ \\ \not{b}^{|w|-1} \end{matrix} \right\} \$$

M next goes into a cycle in each step of which it imitates a rewriting of the leftmost member of I on T2. At the beginning of some step, let the contents of T2 be

$$xY\beta\not{b}^{\ell} \qquad \text{where} \quad x \in \Sigma^*, \quad Y \in I, \quad \beta \in \underline{V}^*.$$

We refer to Y and to the point below it on T3 as the derivation point. The portion of T3 lying to the left of the derivation point is the output of T given x as input. Hence, this substring determines the changes to the local production set at the time that Y is rewritten by the grammar. It is therefore possible to "choose" a member of the local production set at random. M either (1) chooses a member, ϕ, of P_o and then scans T3 from its left boundary to the derivation point, to verify that ϕ is not deleted, or (2) scans T3 leftward from the derivation point, choosing an added production, ϕ, at random, and then scans rightward from the point of choice to the derivation point, to verify that ϕ is not subsequently deleted. In either case, if ϕ is deleted, then M rejects.

Let the production so chosen be $A \rightarrow \alpha$, where $A \in I$ and $\alpha \in \underline{V}^+$. The string β is moved $(|\alpha| - |A|)$ tape squares to the right along T2, and α is copied into the region between x and β. T2 then contains

$$x\alpha\beta\not{b}^{\ell'} \qquad \text{where} \quad x \in \Sigma^*, \quad \alpha \in \underline{V}^+, \quad \beta \in \underline{V}^*.$$

If an attempt is made to move β off the right end of T2, then M rejects.

The above cycle is repeated until T2 contains no members of I.

M then compares the contents of T1 and T2 and accepts if and only if they are identical. Since each step of the cycle corresponds to a legal generation step, and since M has previously verified that T would have accepted w, it follows that if w is accepted by M, then $w \in L(G)$. Conversely, if $w \in L(G)$, it has at least one L-restricted derivation which M can imitate. Hence, the language accepted by M is precisely L(G). □

Corollary 3.3.2

The family of L-restricted ECF languages is a proper subset of the family of CS languages.

Proof

The above theorem shows that the L-restricted ECF languages are a subset of the CS. To show that they form a proper subset, we observe that the language used as a counter-example in proving Theorem 3.2.2 was L-restricted. Hence, the family of L-restricted ECF languages is not closed under non-erasing homomorphism. Since the family of CS languages is closed under non-erasing homomorphism, the result follows. □

Remark. Note that the above theorem and corollary are valid if the L-restriction is redefined to assert the weaker condition $|A| \leq K|\alpha|$ for any fixed constant K. Also, they continue to hold if productions $A \rightarrow e$, where $A \in V_N$, are admitted. The proof of the latter assertion involves the following construction: M "guesses" which symbols on T2 will generate the empty string, erases these symbols, records the guesses, and later verifies their legality.

If the L-restriction is completely removed, then the construction used in Theorem 3.3.1 will not yield a recognizer that operates in space n. Indeed, it may be that $(w, \beta) \stackrel{*}{\Longrightarrow} (ww', e)$ with $|\beta| > |w'|$, so

that M will attempt to move β off the right end of T2 and will fail. However, with some modifications to handle erasing rules, the construction will yield a recognizer which operates in space n^2.

Theorem 3.3.3

Let G be an ECF grammar. There exists a constant K such that L(G) can be recognized by a nondeterministic Turing machine M having tape bound Kn^2.

Proof

We use the construction employed in proving Theorem 3.3.1 with certain modifications. Consider the cycle in which M imitates a rewriting of the leftmost member of I on T2. At the beginning of some step, let the contents of T2 be

$$x\, Y\, \beta\, \not{b}^{\ell} \qquad \text{where } x \in \Sigma^*, \quad Y \in I, \quad \beta \in \underline{V}^*.$$

Let the production selected to be used in rewriting be $Y \rightarrow \alpha$. If $\alpha = e$ then M rejects, so that erasing rules are never applied directly.

Instead, M operates as follows. The rewriting step results in

$$x\, \alpha\, \beta\, \not{b}^{\ell'} \qquad \text{where } x \in \Sigma^*, \quad \alpha \in \underline{V}^+, \quad \beta \in \underline{V}^*.$$

The string α is composed of one or more elements of $\underline{V}$, say $\alpha = A_1 \ldots A_n$ where $A_i \in \underline{V}$. Some of the A's may be members of I; M nondeterministically guesses which of these would rewrite to the empty string in a derivation of w by G. M marks these A's specially. Hence, at a given step in the cycle of imitating G, T2 has the form

$$x B_{11}B_{12}\cdots B_{1i_1}C_{11}C_{12}\cdots C_{1j_1}B_{21}B_{22}\cdots B_{2i_2}C_{21}C_{22}\cdots C_{2j_2}$$

$$\cdots\cdot B_{N1}B_{N2}\cdots B_{Ni_N}C_{N1}C_{N2}\cdots C_{Nj_N}\not{b}^{\ell''}$$

where

$i_k, j_k \geq 1 \qquad \forall\ k$

$C_{pq} \in I$ and is marked to indicate a guess that C_{pq} will generate the empty string

$B_{rs} \in \underline{V}$

By the construction used, each B_{rs} will generate at least one terminal. Hence, if the derivation is to produce w, we must have $N \leq |w|$. If N ever exceeds $|w|$ then M rejects.

We can now state the rewriting step more precisely. At the beginning of some step, let the contents of T2 be

$$x Z \beta \not{b}^{\ell} \qquad \text{where } x \in \Sigma^*,\quad Z \in I,\quad \beta \in \underline{V}^*.$$

If Z is a B_{rs} (i.e., predicted not to rewrite to the empty string) then rewriting proceeds as discussed above: some non-erasing rule $Z \rightarrow \alpha$ is applied. If, however, Z is a C_{pq} (i.e., predicted to rewrite to the empty string), then M checks the prediction. This entails determining whether there is a derivation sequence $Z \overset{*}{\Rightarrow} e$ using the local production set at this point. Since no rules can be added to the local production set while rewriting Z to e, this verification requires no additional storage. If the verification fails then M rejects.

The cycle is repeated until T2 contains no members of I. M then compares T1 with T2 and accepts if and only if they are identical. As in Theorem 3.2.1, the language accepted by M is precisely L(G).

To obtain the space bound, consider the complete substring $C_{k1}C_{k2}\ldots C_{kj_k}$, for some k. Since each C_{kt} is predicted to produce the empty string and hence leave the local production set unchanged, we lose no information if duplicate elements are removed. Since checking for duplicate elements requires no additional space, M can remove them during the rewriting step without affecting the space bound. Hence, we assume that $C_{k1}\ldots C_{kj_k}$ contains no duplicates, for all k.

Further, every C_{kt} in such a substring must appear in a production. Either this production is in P_o, or it is in T(w). Hence,

$$|C_{k1}\ldots C_{kj_k}| \leq |T(w)| + K_2$$

where K_2 is the number of distinct elements of I found in the initial production set P_o. Letting K_1 be the maximum number of symbols emitted by the FST for any single input, we have

$$|C_{k1}\ldots C_{kj_k}| \leq K_1|w| + K_2.$$

Hence, the length of all C_{pq}'s at any point in the cycle is bounded by

$$(K_1|w|+K_2)\cdot N \leq (K_1|w|+K_2)|w|.$$

Since all B_{pq}'s generate at least one symbol, the number of B_{pq}'s must be $\leq |w|$. Since each B_{pq} must also appear in a production, the length of all B_{pq}'s at any point is also bounded by

$$(K_1|w|+K_2)|w|.$$

Hence, the number of symbols on T2 at any point is bounded by

$$|w| + 2|w|(K_1|w| + K_2).$$

Therefore, taking $K > 2(K_1 + K_2) + 1$, M can recognize any string w in space $K|w|^2$.

Remark. The result we shall present in Sections 4.3 and 4.5 can be used to obtain an analogous result for recognition by a deterministic Turing machine. We will exhibit a recognition algorithm for a random access machine which runs in time and space n^8. It follows, therefore, that the same algorithm will have polynomial bounds when modified to run on a Turing machine.

3.4 UNDECIDABILITY RESULTS

Although it is possible to decide whether a given string is generated by a given ECF grammar, we show in this section that it is not possible to decide whether an ECF grammar generates any terminal strings whatever. That is, the emptiness problem for ECF grammars is undecidable. This property appears fundamental to ECF grammars. It continues to hold even when a number of strong restrictions are placed on the grammars. In Section 5.1, we will discuss the significance of this result in applying ECF grammars to the description of programming languages.

Theorem 3.4.1

The question $L(G) \stackrel{?}{=} \emptyset$ is undecidable for ECF grammars, even under the following restrictions:

(a) the L-restriction holds,

(b) no productions are ever deleted,

(c) a production generated by the FST may have only terminal symbols on its right-hand side.

Proof

For any Turing machine M and any initial configuration C, there exists (effectively) an ECF grammar $G = \mathcal{G}(M, C)$ such that $L(G) \neq \emptyset$ if and only if M halts when started in configuration C. From the undecidability of the halting problem for Turing machines follows the undecidability of the emptiness problem for ECF grammars. The construction is as follows.

Let $M = (K, \Sigma, \Gamma, \delta, q_o, F)$ be a Turing machine. Assume $K \cap \Gamma = \emptyset$, so that an instantaneous description may be represented unambiguously by a string $\$ \alpha q \beta \$$ where $\alpha, \beta \in \Gamma^*$, $q \in K$, and "$" and "$" are special symbols which delimit the instantaneous description.

Let an initial configuration of M be $C_o = \alpha_o q_o \beta_o$. The corresponding ECF grammar (for M applied to C_o) is given by $G = (V, \Sigma', P_o, X, T)$ where P_o is given by:

$$X \rightarrow CN$$

$$C \rightarrow \$ \alpha_o q_o \beta_o \$$$

$$N \rightarrow CN .$$

The finite state transducer, T, is defined so as to map each input ID $\$ \alpha q \beta \$$ into an output production $[\![C \rightarrow \$ \alpha' q' \beta' \$]\!]$ such that $\alpha q \beta \vdash_M \alpha' q' \beta'$. That this mapping can be carried out using finite memory is clear: In obtaining a successor ID, the state symbol is moved at most one square in some direction, and at most one other symbol is changed.

Hence, a derivation of G has the form

$$X \Longrightarrow CN \Longrightarrow \S\,\alpha_0\, q_0\, \beta_0\, \$\, N$$
$$\Longrightarrow \S\,\alpha_0\, q_0\, \beta_0\, \$\, C\, N$$
$$\Longrightarrow \S\,\alpha_0\, q_0\, \beta_0\, \$\ \ \S\,\alpha_1\, q_1\, \beta_1\, \$\, N$$
$$\overset{*}{\Longrightarrow} \S\,\alpha_0\, q_0\, \beta_0\, \$ \ldots \S\,\alpha_n\, q_n\, \beta_n\, \$\, N$$

where, for all j $(0 < j \leq n)$,

$C_j \equiv \alpha_j\, q_j\, \beta_j$ either is the immediate successor (under a derivation of M) of C_{j-1}, or is equal to C_k for some k $(0 \leq k < j)$.

The FST has one additional function. If it ever emits a production $[\![\, C \rightarrow \S\,\alpha\, q\, \beta\, \$ \,]\!]$ where q is a final state of M, then it also emits a production $[\![\, N \rightarrow h \,]\!]$. Hence, if the Turing machine M ever reaches a final state, there will be at least one derivation of G having the form

$$X \overset{*}{\Longrightarrow} \S\,\alpha_0\, q_0\, \beta_0\, \$ \ldots \S\,\alpha_n\, q_n\, \beta_n\, \$\, h$$

so that $L(G) \neq \phi$. Since this is the only way N can ever be rewritten directly to a terminal string, the converse holds. □

Corollary 3.4.2

The following are undecidable for ECF grammars:

(a) whether L(G) is context-free, finite, or regular,

(b) whether G is L-restricted,

(c) whether derivations of G involve no deletion of productions,

(d) given k, whether G is RLB-k.

Proof

Clearly, all the above properties hold for any ECF grammar whose language is empty. Also, for each of the above, there exists an ECF grammar G such that the property in question does not hold. Example 2.3.1 gives a grammar G such that L(G) is not context-free,

finite, or regular; also, G is not RLB-k, for any k. Example 2.3.3 gives a grammar which is not L-restricted. It is easy to construct a grammar such that its derivations may involve deletion of productions.

Given the desired grammar $G = (V, \Sigma, P_o, X, T)$, the proof is identical for all four of the assertions. Let $G' = (V', \Sigma', P'_o, X', T')$ be the grammar used in the proof of Theorem 3.4.1. We construct a new grammar $G'' = (V'', \Sigma'', P''_o, X'', T'')$ from G and G' such that the property in question holds iff $L(G'') = \phi$.

Assume $V \cap V' = \phi$. Let $V'' = V \cup V' \cup \{m\}$ and let $\Sigma'' = \Sigma \cup \Sigma' \cup \{m\}$ where m is a new symbol. Let $P''_o = P_o \cup P'_o$. Let $X'' = X'$. T'' is constructed from T and T' as follows. T'' contains an image of T and a modified image of T'. The start state of T'' is that of T', so that T'' initially imitates T'. The modification is that where T' would emit the terminal production $[\![N \rightarrow h]\!]$, T'' emits the production $[\![N \rightarrow m X]\!]$, where m is a special marker and X is the start symbol of G. T'' then goes into a special state in which the only acceptable input is m; if m is found as the next symbol, then T'' enters the start state of T and subsequently imitates T.

By construction,

$$L(G'') = \{w' m w \mid w' \in L(G'), \quad w \in L(G)\}.$$

Hence, if $L(G'') \neq \phi$ then the property in question does not hold. If $L(G'') = \phi$ then the property clearly holds. Also, $L(G'') = \phi$ iff $L(G') = \phi$. Since the latter question is undecidable, the property is undecidable. □

3.5 RESTRICTED CASES

We turn to consideration of the classes of languages produced by imposing various restrictions on ECF grammars. Each restriction limits in some fashion the form of a new production. By studying the classes of languages thereby produced, we obtain a more precise understanding of the generative power of ECF grammars.

In Section 3.1, the notion of a rule length bounded ECF grammar was defined, and it was shown that such a grammar generates only a context-free language. Another restriction which leads to producing only the context-free languages is obtained by considering ECF grammars in which all productions emitted by the FST are deletion rules, i.e., of the form $[\![A \not\rightarrow \alpha]\!]$. Since rules which delete productions not in P_o may be ignored, such a grammar can be imitated by a push-down automaton. Hence, its language is context-free.

A more interesting type of restriction is that yielding families of languages which both:

(a) properly contain the context-free,

(b) are properly contained in the ECF.

We consider two restrictions which have this property:

(1) grammars in which new productions have only terminal symbols on their right-hand side,

(2) grammars for which there is a bound on the number of new productions.

Each of these demonstrates a facility of ECF grammars which can be omitted only with the loss of generative power.

Definition 3.5.1. A derivation is I-restricted if $\forall$ productions ϕ used in the derivation, either $\phi \in P_o$ or $\phi = A \rightarrow \alpha$ where $\alpha \in \Sigma^*$. An ECF grammar is I-restricted if $\forall$ $w \in L(G)$, $\exists$ some I-restricted terminal derivation of w.

Definition 3.5.2. Let $G = (V, \Sigma, P_o, X, T)$ be an ECF grammar and let $w \in L(G)$. The string w is rule number bounded with constant k (RNB-k) if T(w) has no more than k substrings which can be interpreted as productions being added (i.e., of the form "$[\![A \rightarrow \alpha]\!]$").
A grammar G is RNB-k if $\forall$ $w \in L(G)$, w is RNB-k.

Since the grammar of Example 2.3.1 is I-restricted and RNB-1, it follows that the family of I-restricted languages and the family of RNB-k languages (for any $k \geq 1$) each properly contain the context-free.

Theorem 3.5.1

The family of I-restricted ECF languages is a proper subset of the family of ECF languages.

Proof

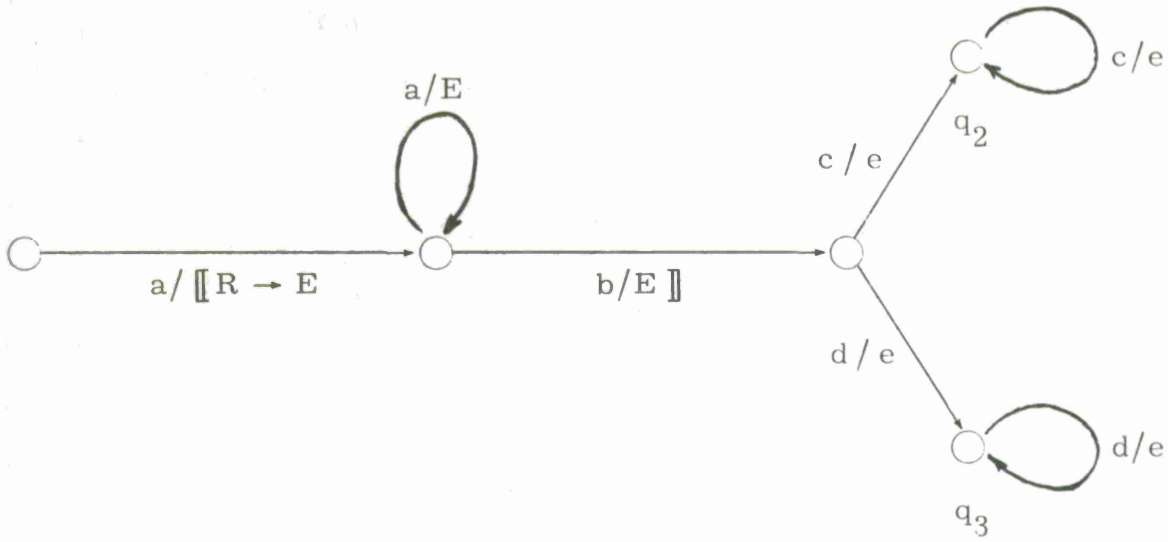
Consider $L = L_1 \cup L_2$, where

$$L_1 = \{a^n b\, c^{(n+1)m} \mid n \geq 1,\ m \geq 2\},$$
$$L_2 = \{a^n b\, d^{(n+1)m} \mid n \geq 1,\ m \geq 2\}.$$

An ECF grammar which generates L is as follows. The initial production set is:

$$X \rightarrow A\ b\ R\ N$$
$$A \rightarrow a\ A \mid a$$
$$N \rightarrow R\ N \mid R$$
$$E \rightarrow c \mid d\ .$$

The finite state transducer is given by:



where $F = \{q_2, q_3\}$. Note that this grammar is not I-restricted, nor can it be modified to be so.

Let $G = (V, \Sigma, P_o, X, T)$ be any ECF grammar, not necessarily the above, whose language is L. We shall demonstrate that G is not I-restricted.

Clearly, L is not context-free. Hence, G is not rule length bounded. Further, for any k $\exists$ at least two strings $w_1 \in L_1$ and $w_2 \in L_2$ which are not RLB-k. Indeed, suppose the contrary. Then $\exists$ k such that either L_1 or L_2 is RLB-k, say L_1. Hence, L_1 is context-free. But since L_1 can be mapped by a homomorphism into the language of Example 2.3.1, this is impossible. Let $w_1 = a^{\ell_1} b c^{m_1}$ and let $w_2 = a^{\ell_2} b d^{m_2}$.

Let the number of states in T be s and let the maximum length of output emitted by T for any single input symbol be N. Let the length of the longest production in P_o be p_m. Let $k > \text{maximum } (p_m, 4N \cdot (s+1))$.

Let $\Pi_1 = \pi_{10}, \ldots, \pi_{1n_1}$ and $\Pi_2 = \pi_{20}, \ldots, \pi_{2n_2}$ be derivations of w_1 and w_2, respectively, using productions ϕ_1 and ϕ_2 such that $|\phi_1| > k$ and $|\phi_2| > k$.

Since $|\phi_i| > p_m$, ϕ_i must have been generated by T, for $i = 1, 2$. Since $|\phi_i| > 4N \cdot (s+1)$, ϕ_i must have been output while the b was read in. Indeed, otherwise ϕ_i would be the output corresponding to a string of a's, c's, or d's. However, this is impossible, for given only a single input, T cannot generate $N \cdot (s+1)$ symbols without being in a loop. If in a loop, then T cannot output a capping "]]" to terminate the production. (This is essentially the same argument as used in the proof of Theorem 3.1.4.) Hence, if $\phi_1 \in P_{\pi_{1j}}$ for any j then $\pi_{1j} = (a^s b c^t, \gamma)$ for some integers s, t and some $\gamma \in \underline{V}^*$. Similarly, if $\phi_2 \in P_{\pi_{2j'}}$ then $\pi_{2j'} = (a^u b d^v, \delta)$ for some u, v, and δ.

For convenience of notation, let $f_1 \equiv c$ and $f_2 \equiv d$, so that f_i can denote either c or d. Let $a^{N_i} b f_i^{N_i!}$ be the substring of w_i which is mapped by T into $[\![\phi_i]\!]$. Let $[\![\phi_i]\!] = \psi_i \cdot \psi_i^b \cdot \psi_i!$ where ψ_i, ψ_i^b, and $\psi_i!$ are the images under T of a^{N_i}, b, and $f_i^{N_i!}$, respectively. Since $[\![\phi_i]\!]$ is capped by a final "]]", $N_i! < s+1$; hence $|\psi_i!| < N(s+1)$. Since $|\psi_i| + |\psi_i^b| + |\psi_i!| = |\phi_i| > 4N(s+1)$, we have $|\psi_i| > 2N(s+1)$. Let ψ_i° be the first $2N(s+1)$ symbols of ψ_i. The only input to T up to the end of ψ_i° is a's; since T is deterministic, $\psi_1^\circ = \psi_2^\circ$. Further, since the productions ϕ_1 and ϕ_2 are used in the derivation, they contain no member of $\underline{V}$ whose length exceeds $N(s+1)$. Hence,

$$\psi_1^\circ = \psi_2^\circ = [\![A \rightarrow \alpha\eta \qquad \text{where } A \in I, \quad \alpha \in V^+, \quad \eta \in V^*.$$

Therefore

$$\phi_1 = A \rightarrow \alpha\beta_1$$
$$\phi_2 = A \rightarrow \alpha\beta_2 \qquad \text{where } A \in I, \quad \alpha \in \underline{V}^+, \quad \beta_i \in \underline{V}^*.$$

Suppose G were I-restricted. Then $\phi_i = A \rightarrow \alpha\beta_i$ where $\alpha \in \Sigma^+$. Since Π_1 uses ϕ_1 and Π_2 uses ϕ_2, it follows that w_1 and w_2 must have the forms

$$w_1 = a^{k_1} b w_1' \alpha \beta_1 w_1''$$

$$w_2 = a^{k_2} b w_2' \alpha \beta_2 w_2''$$

where $\alpha \neq e$. But by hypothesis,

$$w_1 = a^{\ell_1} b c^{m_1}$$

$$w_2 = a^{\ell_2} b d^{m_2}.$$

Hence, $c^{m_1} = w_1' \alpha \beta_1 w_2''$ and $d^{m_2} = w_2' \alpha \beta_2 w_2''$ which is impossible, since α must consist of c's in one case and d's in the other. Contradiction.

We conclude that if L = L(G), then G is not I-restricted. □

Theorem 3.5.2

For each k, the family of RNB-k ECF languages is a proper subset of the family of ECF languages.

Proof

Consider the language
$L = \{abac\,a^2ba^2c\,a^3ba^3c \ldots a^nba^nc \mid n \geq 1\}$. By the theorem of Bar-Hillel, Perles, and Shamir [Hop69] this language is not context-

free. It can, however, be generated by an ECF grammar. Let $G = (V, \Sigma, P_o, X, T)$, where $\Sigma = \{a, b, c\}$, $V_N = \{X, Y, Y', B, B'\}$, and P_o is given by:

$$X \rightarrow Y'Y'$$

$$Y' \rightarrow a\,B$$

$$B \rightarrow b \mid c\,YY \mid c$$

$$B' \rightarrow b \mid c\,Y'Y' \mid c\ .$$

The finite state transducer is

$$T = (K, \Sigma, \Delta, \delta, \lambda, q_1, F),$$

where

$$K = \{q_1, q_2, \ldots, q_8),$$

$$\Delta = V_N \cup \{[\![\ ,\ ]\!],\ \rightarrow,\ \not\leftarrow\},$$

and

$$F = \{q_1, q_5\}.$$

The transition and output functions are as follows:

$$\delta(q_1, a) = q_2 \qquad \lambda(q_1, a) = [\![\ Y \rightarrow aa$$

$$\delta(q_2, a) = q_2 \qquad \lambda(q_2, a) = a$$

$$\delta(q_2, b) = q_3 \qquad \lambda(q_2, b) = B']\!]$$

$$\delta(q_3, a) = q_4 \qquad \lambda(q_3, a) = [\![\ Y' \not\leftarrow a$$

$$\delta(q_4, a) = q_4 \qquad \lambda(q_4, a) = a$$

$$\delta(q_4, c) = q_5 \qquad \lambda(q_4, c) = B]\!]$$

$$\delta(q_5, a) = q_6 \qquad \lambda(q_5, a) = [\![\ Y' \rightarrow aa$$

$$\delta(q_6, a) = q_6 \qquad \lambda(q_6, a) = a$$

$$\delta(q_6, b) = q_7 \qquad \lambda(q_6, b) = B \,]\!]$$

$$\delta(q_7, a) = q_8 \qquad \lambda(q_7, a) =]\![Y \nwarrow a$$

$$\delta(q_8, a) = q_8 \qquad \lambda(q_8, a) = a$$

$$\delta(q_8, c) = q_1 \qquad \lambda(q_8, c) = B' \,]\!].$$

The FST alternates between two activities: (1) mapping a^i into a production with $a^{i+1}B$ as its right-hand side, (2) mapping a^i into the deletion of a production with a^iB as its right-hand side. Hence, each substring a^iba^ic, when mapped by the FST, first creates a production which will allow generation of its successor and then deletes the production which is used in generating itself.

Let $G = (\overline{V}, \overline{\Sigma}, \overline{P}_o, \overline{X}, \overline{T})$ be an arbitrary ECF grammar such that $L(\overline{G}) = L$. We claim that for any given integer k, $\overline{G}$ is not RNB-k. Suppose the contrary, i.e., $\exists$ k such that $\overline{G}$ is RNB-k. Let $w_n = abac\ a^2ba^2c \ldots a^nba^nc$. Consider $\overline{T}(w_n)$. Since $\overline{G}$ is RNB-k, then for any n, $\overline{T}(w_n)$ contains at most k substrings which can be interpreted as added productions. That is, the greatest number of such substrings which can appear in any $\overline{T}(w_n)$ is some $s \leq k$. Let this be attained for the string w_m. For all $n > m$, the deterministic action of $\overline{T}$ guarantees that $\overline{T}(w_n)$ contains these productions; hence, it contains no <u>other</u> added productions. Therefore, $\{w_n \mid n \geq 1\}$ is rule length bounded. Since $L = \{w_n \mid n \geq 1\}$, L is context-free. As this is impossible, we conclude that $\overline{G}$ is not RNB-k for any k. □

Collecting the above two theorems and the remarks which preceded them, we have the following:

Theorem 3.5.3

The families of I-restricted and RNB-k ECF languages each

(a) properly contain the family of context-free languages,

(b) are properly contained in the family of ECF languages.

Remark. We note that these two restricted classes of ECF grammars share another trait: an undecidable emptiness problem. The assertion has been proved for the I-restricted case in Theorem 3.4.1; the proof for the RNB-k case is given by the following theorem.

Theorem 3.5.4

The emptiness problem is undecidable for the class of RNB-k grammars, for any $k \geq 2$.

Proof

Let $\{(\alpha_1,\beta_1), (\alpha_2,\beta_2), \ldots, (\alpha_n,\beta_n)\}$ where $\alpha_i,\beta_i \in \bar{\Sigma}^+$ be a Post correspondence problem. We construct an RNB-2 grammar whose language is non-empty if and only if the correspondence problem has a solution.

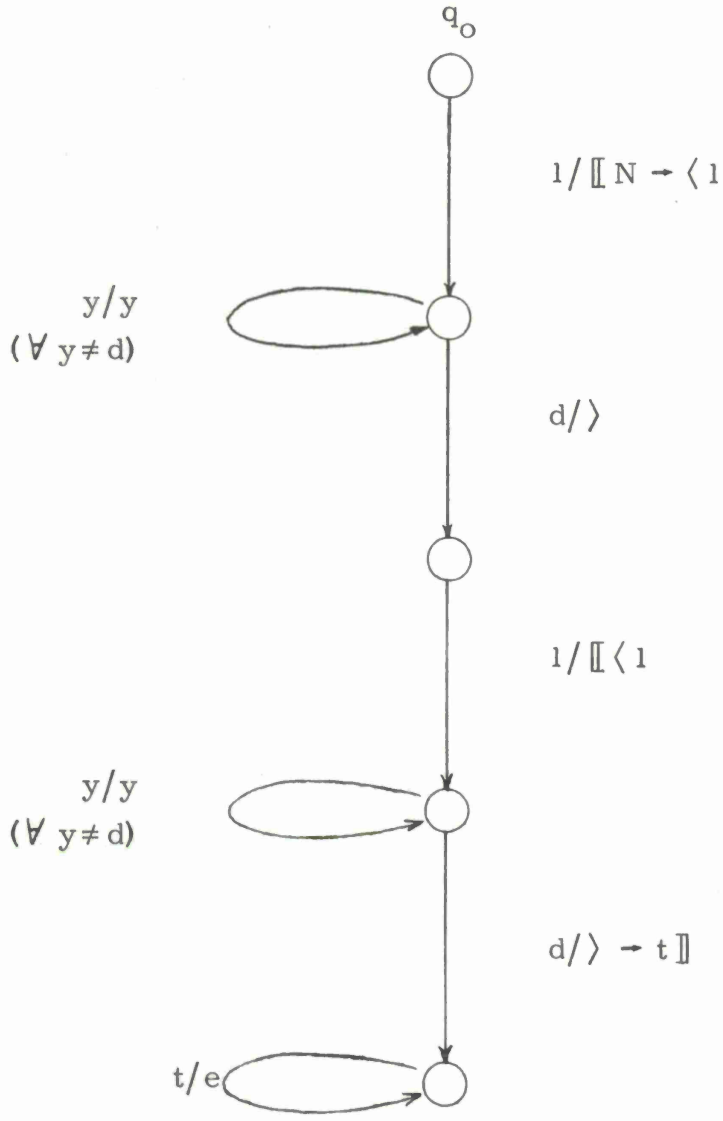
The initial production set P_o is given by the schema

$$X \rightarrow A\,d\,B\,d\,N$$

$$A \rightarrow 1\,0^i A\,\alpha_i \;\mid\; 1\,0^i c\,\alpha_i \qquad i = 1,\ldots,n$$

$$B \rightarrow 1\,0^i B\beta_i \;\mid\; 1\,0^i c\beta_i \qquad i = 1,\ldots,n$$

where $0, 1, c, d$ are new terminal symbols not members of $\bar{\Sigma}$. The FST is specified by its state transition diagram



Let L_A be the context-free language generated by taking the start symbol to be A and using only productions in P_o with A as their left-hand side; let L_B be analogously defined. A derivation of the ECF grammar must have the form

$$(e,X) \Rightarrow (e,A\,d\,B\,d\,N)$$

$$\overset{*}{\Rightarrow} (w_A\,d,B\,d\,N) \qquad \text{where } w_A \in L_A$$

at which point the local production set is

$$P_o \cup \{N \rightarrow \langle w_A \rangle\}.$$

The derivation must continue

$$\stackrel{*}{\Rightarrow} (w_A\, d\, w_B\, d, N) \qquad \text{where } w_A \in L_A, \quad w_B \in L_B$$

at which point the local production set is

$$P_o \cup \{N \rightarrow \langle w_A \rangle, \langle w_B \rangle \rightarrow t\}.$$

Hence, the derivation must continue

$$\Rightarrow (w_A\, d\, w_B\, d, \langle w_A \rangle).$$

This terminates iff $w_A = w_B$ iff $L_A \cap L_B \neq \phi$ iff the correspondence problem has a solution. □

3.6 RELATION TO OTHER GENERALIZATIONS OF CONTEXT-FREE GRAMMARS

To date, at least six generalizations of context-free grammars have been proposed: scattered context [Greib68], table grammars [Whit68a], [Whit68b], [Whit68c], [Whit69], indexed grammars [Aho68], macro grammars [Fisch68], programmed grammars [Ros69], and grammars with control sets [Gins68b]. To complicate matters, some of these have two or more subfamilies. With the notable exception of [Fisch68] (which establishes definite relations to [Aho68], little work has been done in determining the hierarchy of these various models. We shall not attempt to undertake such a study in this paper. Instead, we shall relate the family of ECF grammars to what we feel is the most significant family above: the indexed languages.

These are of special interest, for they are generated by a number

of apparently unrelated formal systems: indexed grammars, OI macro grammars, nested stack automata [Aho69a], and pushdown automata in which the stack elements are themselves stacks [Aho69b]. This suggests that the family embodies some central, machine-independent notion and hence will be of particular importance in the study of formal languages.

Having thus justified a comparison with the indexed languages, it is unfortunate – but nonetheless of interest – to assert:

Theorem 3.6.1

The families of ECF languages and indexed languages are incommensurable; i.e., neither family is a subset of the other.

Proof

Fischer [Fisch68] shows that $L = \{a^n \mid n \text{ is non-prime and } \geq 2\}$ is a basic macro language (see [Fisch68] for definitions). Hence, L is an OI macro language and equivalently is an indexed language. In view of Corollary 3.1.5, L is not ECF.

To show the converse, we construct an ECF grammar which "imitates" a universal Turing machine and show that the language it generates cannot be an indexed language.

Let $M = (K, \Sigma, \Gamma, \delta, q_0, F)$ be a universal Turing machine. As noted in the proof of Theorem 3.4.1, an instantaneous description of M can be unambiguously represented by a string $\S\, \alpha\, q\, \beta\, \S$ where $\alpha, \beta \in \Gamma^*$, $q \in K$, and $\S$ and $\S$ are special delimiters not in Γ or K.

Consider the ECF grammar used in the proof of Theorem 3.4.1 modified so that its initial production set is

$$X \rightarrow \S S q_o S \S N$$

$$S \rightarrow e \mid aS \qquad \forall\, a \in \Sigma$$

$$N \rightarrow CN$$

A derivation in the new grammar begins

$$X \Rightarrow \S S q_o S \S N$$

$$\overset{*}{\Rightarrow} \S \alpha q_o \beta \S N \qquad \text{where } \alpha, \beta \in \Gamma^*$$

$$\Rightarrow \S \alpha q_o \beta \S C N$$

Hence, a derivation generates an arbitrary initial configuration and then imitates the sequence of instantaneous configurations produced by M when started on this initial configuration. The language generated by the ECF grammar is

$$L = \{C_o C_1 \ldots C_n h \mid C_o \text{ is an initial configuration of } M$$

$$\text{and } \forall\, j = 1, \ldots, n \text{ either } C_{j-1} \vdash_M C_j \text{ or}$$

$$C_j = C_k \text{ for some } k \ (0 < k < j)\}.$$

Hence, a string in L is an encodement of a halting computation of M.

Consider the gsm g which maps each symbol into that symbol, up to and including the first "§" it encounters and thereafter maps each symbol into the empty string. Then

$$g(L) = \{C_o \mid M \text{ halts when applied to initial configuration } C_o\}.$$

Since the halting problem is undecidable, g(L) is not recursive.

Suppose L were an indexed language. Since the family of indexed

languages is closed under gsm mappings [Aho68], g(L) would be an indexed language. However, since the indexed languages are recursive, this is impossible. Contradiction. □

Remark. For two other families – the scattered context languages and the cfpg programmed grammars – it is possible to exhibit languages which are not ECF. The converse questions, however, are open.

Section 4. PARSING

4.1 MOTIVATION

We now turn to the problem of parsing strings generated by an ECF grammar. As claimed in Section 3.3, we will demonstrate that legal strings can, in fact, be recognized, i.e., that the ECF languages are recursive. However, the theoretical question is of only secondary interest. If ECF grammars are to be used to specify programming languages, we require not merely a recognizer but a parser. Further, the parse algorithm must be sufficiently economical in time and space to be of practical utility. The algorithm we will exhibit has this property.

4.2 AN ALTERNATE FORMALISM FOR DERIVATIONS

In Section 2.2, after defining an ECF derivation, we noted that an alternate definition exists. As this alternate definition is far more efficient for computational purposes, it is a preferable one to use in a discussion of efficient parsing.

Definition 4.2.1. Let $G = (V, \Sigma, P_0, X, T)$ be an ECF grammar. A configuration of G is defined to be an element of $(\Sigma^*, \underline{V}^*, K_T, \Delta_T^*, S)$ where K_T is the state set of T, where Δ_T is the output vocabulary of T, and where S is the set of possible production sets over V.

The transition between a configuration ψ and a possible successor ψ' is denoted by $\psi \vdash \psi'$ and is defined as follows.

(1) If $(A \rightarrow \alpha) \in P$, then $(w, A\beta, q, x, P) \vdash (w, \alpha\beta, q, x, P)$,

(2) If $a \in \Sigma$, then $(w, a\beta, q, x, P) \vdash (wa, \beta, q', x', P')$ where $q' = \delta(q, a)$, $P' = \mathbb{P}(P, x \cdot \lambda(q, a))$, and where x' is obtained as follows. Let $y = x \cdot \lambda(q, a)$. If y does not contain the symbol "$\rrbracket$", then let $x' = y$. Otherwise, write $y = y_1 \cdot \rrbracket \cdot y_2$ where y_2 does not contain "$\rrbracket$"; let $x' = y_2$.

Let $\vdash^{m}$ and $\vdash^{*}$ denote the m-fold closure and transitive closure of $\vdash$, defined in the usual fashion.

The notion of configuration is related to that of instantaneous description (cf. Section 2.2) as follows. If $\pi = (w, \gamma)$ is an ID, an equivalent configuration is given by $\psi = (w, \gamma, \delta(q_o, w), y, P_\pi)$ where y is that substring of $T(w)$ which is right of the rightmost instance of "$\rrbracket$". The configuration ψ differs from the ID π in that it explicitly carries (1) the local production set, and (2) part of the information needed to compute the local production set of a successor configuration. Generation expressed as a sequence of configurations simply avoids the total recomputation of P_π at every step.

Theorem 4.2.1

For any ECF grammar, $(e, X) \Longrightarrow^{m} (w, \gamma) = \pi$ if and only if $\exists$ y such that $(e, X, q_o, e, P_o) \vdash^{m} (w, \gamma, \delta(q_o, w), y, P_\pi)$.

Proof

Obvious, by induction on m. □

In view of the equivalence of ID's and configurations, we will be somewhat loose in our notation. We will use the latter in obtaining time bounds and revert to the former when concise notation is desired.

4.3 AN ADAPTATION OF EARLEY'S ALGORITHM

Of those parse algorithms which handle the entire family of context-free languages, Earley's [Earl68] seems to be the best. It matches the best known time result, n^3, for the general case. For a number of subfamilies on which a special algorithm will run faster (e.g., Kasami's time-n^2 algorithm for unambiguous grammars [Kas67] and Knuth's time-n algorithm for LR(k) grammars [Knu65]), Earley's algorithm runs at the rate of the special case algorithm. Further, it attains the faster rate automatically, without being instructed that the language in question falls into a special class.*

For our purposes, Earley's algorithm has another useful trait: it places no restrictions whatever on the grammar. Unlike most algorithms, it correctly handles circular grammars, disconnected grammars, and grammars which generate strings having an infinite number of parses. The results of Section 3.1 demonstrate that most normal forms for context-free grammars (e.g., intermediate constituent form or Greibach normal form) are not normal forms for ECF grammars since these normal forms put bounds on the length of productions. Hence, Earley's algorithm, which does not depend on a normal form and which works correctly on any set of productions it is given, is particularly attractive.

We will discuss how Earley's algorithm may be adapted to the parsing of ECF languages, will prove that the resulting algorithm is valid, and will exhibit time and space bounds. We will assume

---

*This is particularly relevant in view of the undecidability results connected with the above special classes.

familiarity with Earley's algorithm as described in Sections II, IV, V, VI, XIV, and XV of his thesis. Definitions and notation will be close to those of Earley. We will also follow Earley in first specifying a recognition algorithm and then discussing how this can be modified to produce a parse.

An intuitive description of our recognition algorithm can best be given in terms of Earley's. The latter operates on two inputs: a string, $a_1 \ldots a_n$, and a grammar, $G = (V, \Sigma, X, P)$. It scans the string from left to right and as each symbol a_i is scanned, it constructs a state set S_i which represents the condition of the recognition process at that point. S_i is a function of three variables: (1) a_i, (2) the previously constructed state sets, $\{S_k \mid k < i\}$, and (3) the set of productions, P. For context-free grammars, P is constant. To allow the algorithm to recognize ECF strings, we simply make P variable. For each i, $P_i = \mathbb{P}(P_o, T(a_1 \ldots a_i))$ is computed and P_i is used in place of P.

One point has been suppressed in the above paragraph. Earley's algorithm also utilizes a k-symbol look-ahead, where k is any fixed non-negative integer. When processing the input symbol a_i, it considers $a_{i+1} \ldots a_{i+k}$ to eliminate false paths as soon as possible. While most of Earley's algorithm carries over to the extensible case, the look-ahead feature does not. In his algorithm, look-ahead consists of verifying an expectation that after some symbol A has been construed in the input string, the next k symbols must be some given string α. In the ECF case, the production set may change while A is being construed, thereby invalidating the expectation α. Hence, we shall first consider an algorithm which involves no look-ahead, i.e., k = 0. Later,

we will discuss how this algorithm can be further modified to include partial look-ahead.

A second point which requires discussion is the fact that Earley's algorithm requires all input strings to be padded on the right by a distinctive symbol, say "$\dashv$", where $\dashv \notin \Sigma$. This requirement can be satisfied in one of two ways. The recognition algorithm can take its input, $a_1 \ldots a_n$, and concatenate to it the symbol "$\dashv$" as the $(n+1)^{st}$ element. Alternatively, the requirement can be cast as a condition imposed on the grammar: i.e., that the start symbol be a special symbol D_o which appears in a unique production

$$D_o \rightarrow X \dashv$$

and that "$\dashv$" appears in no other production. The two methods are entirely equivalent for all practical purposes. However, the first method would induce clumsy notation in later proofs, for it requires special handling of the pad symbol. Hence, we adopt the latter convention. That D_o and $\dashv$ appear in only one production of P_o may be imposed as part of the definition of ECF grammars; to insure that they appear in no new production, we require that D_o and $\dashv$ are not members of the output vocabulary of T. We stress, however, that this convention is made for convenience only and involves no loss of generality.

We now turn to a precise description of the recognition algorithm. For each symbol a_i scanned, two actions are taken: (1) the local production set, P_i, is updated, (2) the state set, S_i, is computed. The former can be performed by a slight modification of the technique described in Section 4.2.

Let $\psi = (w, \beta, q, x, P)$ with successor $\psi' = (wZ, \beta', q', x', P')$. We observe that:

(1) if $Z = e$, then $P' = P$,

(2) otherwise, P' depends only on Z, q, x, and P.

Hence, to compute the local production set, P', we need record only the 3^{rd}, 4^{th}, and 5^{th} components of a configuration.

For any string $a_1 \ldots a_n$ and any i $(0 \leq i \leq n)$, define a <u>string state</u> Q_i as follows:

(1) $Q_o = (q_o, e, P_o)$,

(2) for $(1 \leq i \leq n)$, let $Q_i = (q_i, x_i, P_i)$, where

$q_i = \delta(q_{i-1}, a_i)$

$P_i = \mathbb{P}(P_{i-1}, x_{i-1}\, \lambda(q_{i-1}, a_i))$

x_i is that substring of $x_{i-1} \cdot \lambda(q_{i-1}, a_i)$ which is to the right of the rightmost instance of "]]".

If $Q_i = (q_i, x_i, P_i)$ is a string state of $a_1 \ldots a_n$, then P_i is the local production set for $a_1 \ldots a_i$. Note that we may view the above definition as a procedure for computing P_i. As each symbol, a_i, of input is read, Q_i is computed from a_i and Q_{i-1}.

This specifies P_i as a set of productions. It is useful to assume that these productions are indexed from 0 up to some N_i. We may assume indexing of the initial productions with $\phi_o = D_o \rightarrow X \dashv$. As new productions are added in forming P_i, new index numbers are used. When a production, say ϕ_j, is deleted, its index number, j, is tagged, signifying that the production is inactive. If such a deleted production is added again, the tag is removed.

Another useful notation is to denote the p^{th} production as

$$\phi_p = D_p \rightarrow C_{p1} C_{p2} \cdots C_{p\bar{p}}$$

where $D_p, C_{pi} \in \underline{V}$ for $i = 1, \ldots, \bar{p}$.

Having specified the computation of P_i, we can describe the recognition algorithm itself.

Definition 4.3.1. A state is a triple of integers (p, j, f). A state set is an ordered set of states. A state is added to a state set by placing it last in the ordered set, unless it is already a member.

Algorithm 4.3.1 (ECF Recognizer)

This is a function, RECF, of two arguments: an ECF grammar G and a terminal string $a_1 \ldots a_n$. It has value true or false (accept or reject) and is computed as follows:

Let S_i be empty $(1 \leq i \leq n)$.

Let P_o be as specified in G.

Let $S_o = \{(0, 0, 0)\}$.

Let $Q_o = (q_o, e, P_o)$.

Let $i = 1$ and go to LOOP.

LOOP:

Process the states of S_i in order, performing one of the following three operations on each state $s = (p, j, f)$:

1. (Predictor) If $j \neq \bar{p}$ and $C_{p(j+1)} \in I$, then $\forall\ \phi_q \in P_i$ such that $D_q = C_{p(j+1)}$, add $(q, 0, i)$ to S_i.
2. (Scanner) If $j \neq \bar{p}$ and $C_{p(j+1)} \in \Sigma$, then if $C_{p(j+1)} = a_i$, then add $(p, j+1, f)$ to S_{i+1}.

3. (Completer) If $j = \bar{p}$, then for each $(q, \ell, g) \in S_f$ (after all states have been added to S_f) such that $D_p = C_{q(\ell+1)}$, add $(q, \ell+1, g)$ to S_i.

If S_{i+1} is empty, then reject.

If $i = n-1$, $S_n = \{(0, 2, 0)\}$, and $\exists\ q \in F$ such that $Q_i = (q, x_i, P_i)$, then accept.

Otherwise, let $i = i+1$, let Q_i be computed as described above, let P_i be its third component, and go to LOOP. end

Comparing this to Earley's recognizer, it will be seen that this differs from the latter only in the following respects. (1) In the predictor, the production set used is variable. (2) Earley's look-ahead computation via his function H_k is absent. (3) The last step of the main loop involves computing the local production set P_i.

4.4 A TIME BOUND

In assessing time and space usage of an algorithm, two conditions should be considered: (1) expected usage in the normal case, (2) bounds for the worst case. Note that these may differ greatly. In this section, we discuss the latter. Specifically, we seek a time bound, for since each step of the algorithm uses at most a constant amount of space, a space bound is obtained directly from a time bound.

To obtain such bounds, one must consider an implementation and a machine model on which the implementation is based. We agree with a contention made by Earley that the most significant properties of real computers are most accurately represented not by a Turing machine,

but by a random access machine. This model has an unbounded number of registers containing non-negative integers and referenced (addressed) by successive non-negative integers. It is assumed that some distinguished register holds the constant 0. Primitive operations on these registers are: (1) copying the contents of one register into another, (2) comparing the contents of two registers, (3) adding or subtracting the constant 1 from the contents of a register ($0 - 1 \equiv 0$). A register may be referenced either directly or indirectly; i.e., its address is the contents of a directly referenced register. Referencing by successive integers allows immediate access to elements of structures which behave like arrays. Indirect addressing allows use of list processing techniques.

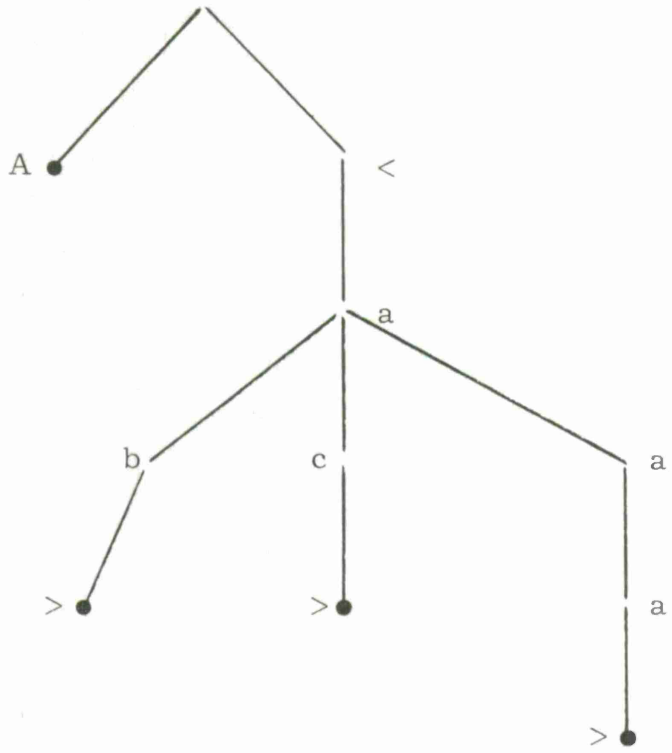
Note that this is a very powerful machine model. For example, such a machine can compute any recursively enumerable set, even if equipped with only three registers. However, such computations involve unrealistic amounts of time and Gödelizations which make the register contents unrealistically large. For those algorithms with which we shall be concerned, time and the magnitude of register contents will be more reasonable.

We begin by considering an implementation of the procedure which computes, for each stage of the scanning, the string state Q_i with its local production set P_i. Let $a_1 \ldots a_i$ be the substring scanned at some point. Let $\delta_i = \{A \mid (A \rightarrow \alpha) \in P_k \text{ for } k \leq i\} \cup$

$$\{B \mid (A \rightarrow \alpha B \beta) \in P_k \text{ for some } \alpha, \beta \in \underline{V}^* \text{ and } k \leq i\}.$$

δ_i is maintained as a tree structure and is updated for each input symbol scanned. For example, the set of intermediate symbols

$\{A, \langle ab\rangle, \langle ac\rangle, \langle aaa\rangle\}$ would be represented by the structure:



Updating S involves tracing down branches and possibly adding new ones.

This structure serves as a symbol table. All instances of members of I are replaced by pointers into S. Hence, aside from the computation required to maintain S and to perform table lookup, the implementation can be carried out as if an infinite set of symbols were available.

For each $A \in I_i$, those productions which have A as left-hand side are kept in a tree structure similar to the symbol table. For each production, status (active or inactive) and length are recorded. The production tree is updated as each input symbol is scanned by the following procedure.

Let the string scanned at some point be $a_1 \ldots a_{i-1}$, with string state $Q_{i-1} = (q_{i-1}, x_{i-1}, P_{i-1})$, where P_{i-1} is the symbol tree. Let $Q_i = (\delta(q_{i-1}, a_i), x_i, P_i)$ where x_i and P_i are given as follows.

(1) If $x_{i-1} \cdot \lambda(q_{i-1}, a_i)$ does not contain the symbol "]]", then no productions have been completed. Hence, $x_i = x_{i-1} \cdot \lambda(q_{i-1}, a_i)$ and and $P_i = P_{i-1}$. This takes at most some constant number of steps.

(2) If $x_{i-1} \cdot \lambda(q_{i-1}, a_i)$ does contain "]]", then there may be one or more productions to process. For each of these, the following is performed.

(a) Its intermediate symbols are encoded into pointers, new symbols in the symbol tree being made when necessary.

(b) The encoded production is looked up in the production tree. If the production is to be added and is not found, then an additional entry is made in the tree. If the production is found in the tree, its status is updated: to <u>active</u> if the production is being added, to <u>inactive</u> otherwise.

This gives a representation of P_i.

To obtain a time bound for this operation, we recall that the number of input symbols processed is i. Hence $| x_{i-1} \cdot \lambda(q_{i-1}, a_i)| \leq ki$, where k is a constant — the maximum length output emitted by the FST for any single input. Therefore, the total time required to perform steps (a) and (b) is bounded by $k_s i$, where k_s is a constant determined by the specific technique chosen to implement the tree structure. It will be useful to suppress such constants and to give time bounds only as they depend on i and hence on the string length n. We will speak of

bounds $k_1 n^N + k_2 n^{N-1} + \ldots + k_{N+1}$ as being of "order n^N", or simply as "n^N". As the procedure for updating P must be applied to each input symbol, the total time required to maintain P is bounded by n^2. (Note that this analysis is quite sloppy; with some care, we can show that the stated procedure requires only time n. However, for the purpose of this section, the result claimed will suffice.)

The other parts of the recognizer are implemented in the same fashion as described by Earley. This implementation is straightforward, with two exceptions.

(1) In the construction of S_i, it is necessary to test each state (p, j, g) to determine if it is already a member of S_i. To save a factor of i in the time required to make this test, a vector of length i is used. The f^{th} entry of this vector points to a list of all states in S_i whose 3^{rd} component is f. To determine whether (p, j, g) is in S_i, it is only necessary to search the g^{th} list.

(2) Erasing rules (i.e., $\bar{p} = 0$) cause some complications to the completer step of the algorithm. Consider applying the completer to the state $(p, 0, i) \in S_i$. It is necessary to consider all $(q, \ell, g) \in S_i$ such that $C_{q(\ell+1)} = D_p$; some of these may yet to be added to S_i. Therefore, it is necessary to maintain a record for each $A \in \mathcal{S}_i$ of whether $(A \rightarrow e) \in P_i$. As each input symbol is scanned, this record is updated along with P. (This increases the time required by at most a constant factor.) For each (p, j, f) added to S_i, if $(C_{p(j+1)} \rightarrow e) \in P_i$, then (p, j+1, f) is added to S_i.

A time bound for the recognizer is obtained as follows. Let $Q_i = (q_i, x_i, P_i)$ be a string state for an initial substring $a_1 \ldots a_i$.

Define:

$d_i = \max_{(1 \leq k \leq i)}$ number of productions in P_k,

$m_i = \max_{(1 \leq k \leq i)}$ length of longest production in P_k,

$\gamma_i = \max_{(1 \leq k \leq i)}$ number of productions in P_k with a common left-hand side.

Clearly, d_i, m_i, γ_i are each of order i (i.e., are bounded by ki for some constant k). In any state set S_i, there are at most $d_i m_i (i+1)$ states. For each of these, one of the following occurs:

(1) Scanner applies. This adds one state to S_{i+1}.

(2) Predictor applies. This adds at most r_i states to S_i.

(3) Completer applies. This adds at most $d_i m_i i$ states to S_i – but note that the completer may be applied to at most $d_i \cdot i$ states.

For each state s added to a state set S_i, it is necessary to check whether s is already a member of S_i. This takes at most $d_i m_i$ steps. Hence, the total time to process S_i is bounded by:

$$[d_i m_i (i+1)(1+r_i) + (d_i i)(d_i m_i i)][d_i i] \simeq d_i m_i i (r_i + d_i i)(d_i i) \simeq i^6 + i^7 \simeq i^7,$$

since all terms are of order i. It will be recalled that the time required to update P_i for each symbol read in is of order i, so that this can be neglected in comparison. Total time for the algorithm is bounded by $C n^8$, where C depends on the grammar but not on the length of input.

4.5 VALIDITY OF THE ALGORITHM

We now turn to a proof that Algorithm 4.3.1 is valid; i.e., $w \in L(G)$ if and only if RECF (G, w) accepts. The proof for the forward assertion carries over rather directly from Earley's proof. In proving the reverse claim, it is found that Earley's proof does not carry over, but that a simplified rendering of his basic idea does work. We begin with the reverse claim.

Definition 4.5.1. Let $w = a_1 \ldots a_n$ be a string of terminals and let $\Pi = \pi_o, \ldots, \pi_k$ be a derivation of w (i.e., $\pi_o = (e, D_o) \overset{*}{\Longrightarrow} (w, \gamma) = \pi_k$ for some γ). The i-states of Π are the triples (p, j, f) such that $\exists\ \gamma \in \underline{V}^*$ and integers i, i_2, i_3, $(0 \leq i_1 < i_2 \leq i_3 \leq k)$ such that:

$$\pi_o \overset{*}{\Longrightarrow} (a_1 \ldots a_f, D_p\gamma) = \pi_{i_1}$$

$$\pi_{i_1} \Longrightarrow (a_1 \ldots a_f, C_{p1} \ldots C_{p\bar{p}}\gamma) = \pi_{i_2}$$

$$\pi_{i_2} \overset{*}{\Longrightarrow} (a_1 \ldots a_i, C_{p(j+1)} \ldots C_{p\bar{p}}\gamma) = \pi_{i_3}\ .$$

Theorem 4.5.1

If $(p, j, f) \in S_i$, then (p, j, f) is an i-state of some derivation of $a_1 \ldots a_i$.

Proof (By induction on the number of states added to any set before (p, j, f) is added to S_i)

Basis. The state (0, 0, 0) is the first state added to S_o. Consider the trivial derivation $\pi_o = (e, D_o) \Longrightarrow (e, X \dashv)$. Letting $\pi_{i_1} = (e, D_o)$, $\pi_{i_2} = (e, X \dashv)$, and $\pi_{i_3} = (e, X \dashv)$, it follows that (0, 0, 0) is an i-state of a derivation of $a_1 \ldots a_i = e$, for $i = 0$.

Induction. For any state added to any state set, one of the following three cases applies.

Case 1. Suppose $(q, 0, i)$ is added to S_i by the predictor acting on (p, j, f).

Since $(p, j, f) \in S_i$, it follows from the induction hypothesis that (p, j, f) is an i-state of $a_1 \ldots a_i$. Hence, $\exists$ a derivation

$$\begin{aligned} \pi_o &\Longrightarrow (a_1 \ldots a_f, D_p \gamma) \\ &\Longrightarrow (a_1 \ldots a_f, C_{p1} \ldots C_{p\bar{p}} \gamma) \\ &\overset{*}{\Longrightarrow} (a_1 \ldots a_i, C_{p(j+1)} \ldots C_{p\bar{p}} \gamma) . \end{aligned}$$

Since the predictor acts on (p, j, f) to obtain $(q, 0, i)$, we have that $C_{p(j+1)} = D_q$ and that $(a_1 \ldots a_i, D_q) \Longrightarrow (a_1 \ldots a_i, C_{q1} \ldots C_{q\bar{q}})$. Hence,

$$\begin{aligned} (a_1 \ldots a_i, C_{p(j+1)} \ldots C_{p\bar{p}} \gamma) &= (a_1 \ldots a_i, D_q C_{p(j+2)} \ldots C_{p\bar{p}} \gamma) \\ &\Longrightarrow (a_1 \ldots a_i, C_{q1} \ldots C_{q\bar{q}} C_{p(j+2)} \ldots C_{p\bar{p}} \gamma) . \end{aligned}$$

Letting $\gamma' = C_{p(j+2)} \ldots C_{p\bar{p}} \gamma$, and collecting selected lines above, we have

$$\begin{aligned} \pi_o &\overset{*}{\Longrightarrow} (a_1 \ldots a_i, D_q \gamma') \\ &\Longrightarrow (a_1 \ldots a_i, C_{q1} \ldots C_{q\bar{q}} \gamma') \\ &\overset{*}{\Longrightarrow} (a_1 \ldots a_i, C_{q(j+1)} \ldots C_{q\bar{q}} \gamma') \end{aligned}$$

for $j = 0$. Hence, $(q, 0, i)$ is an i-state of a derivation of $a_1 \ldots a_i$.

Case 2. Suppose $(q, j+1, g)$ is added to S_i by the completer acting on $(p, \bar{p}, f) \in S_i$ and $(q, j, g) \in S_f$.

Since $(q, j, g) \in S_f$, (q, j, g) is an i-state of $a_1 \ldots a_f$. Hence,

$$\pi_o \overset{*}{\Longrightarrow} (a_1 \ldots a_g, D_q \gamma)$$

$$\Longrightarrow (a_1 \ldots a_g, C_{q1} \ldots C_{q\bar{q}} \gamma)$$

$$\Longrightarrow (a_1 \ldots a_f, C_{q(j+1)} \ldots C_{q\bar{q}} \gamma) .$$

Since $(p, \bar{p}, f) \in S_i$, we have

$$\pi_o \overset{*}{\Longrightarrow} (a_1 \ldots a_f, D_p \gamma')$$

$$\Longrightarrow (a_1 \ldots a_f, C_{p1} \ldots C_{p\bar{p}} \gamma')$$

$$\overset{*}{\Longrightarrow} (a_1 \ldots a_i, \gamma') .$$

Since the completer acts on $(p, \bar{p}, f) \in S_i$ and $(q, j, g) \in S_f$ to produce $(q, j+1, g)$, it follows that $C_{q(j+1)} = D_p$. Hence,

$$\pi_o \overset{*}{\Longrightarrow} (a_1 \ldots a_g, D_q \gamma)$$

$$\overset{*}{\Longrightarrow} (a_1 \ldots a_f, C_{q(j+1)} \ldots C_{q\bar{q}} \gamma)$$

$$= (a_1 \ldots a_f, D_p C_{q(j+2)} \ldots C_{q\bar{q}} \gamma)$$

$$\Longrightarrow (a_1 \ldots a_f, C_{p1} \ldots C_{p\bar{p}} C_{q(j+2)} \ldots C_{q\bar{q}} \gamma)$$

$$\overset{*}{\Longrightarrow} (a_1 \ldots a_i, C_{q(j+2)} \ldots C_{q\bar{q}} \gamma) .$$

Therefore, $(q, j+1, g)$ is an i-state of a derivation of $a_1 \ldots a_i$.

Case 3. Suppose $(p, j+1, f)$ is added to S_{i+1} by the scanner acting on $(p, j, f) \in S_i$. Since (p, j, f) is an i-state of a derivation of $a_1 \ldots a_i$, we have

$$\pi_o \Longrightarrow (a_1 \ldots a_f, D_p \gamma)$$

$$\Longrightarrow (a_1 \ldots a_f, C_{p1} \ldots C_{p\bar{p}} \gamma)$$

$$\Longrightarrow (a_1 \ldots a_i, C_{p(j+1)} \ldots C_{p\bar{p}} \gamma) .$$

Since the scanner adds $(p, j+1, f)$ to S_i, it follows that $C_{p(j+1)} = a_{i+1}$. Hence,

$$(a_1 \dots a_i, C_{p(j+1)} \dots C_{p\bar{p}}\, \gamma) = (a_1 \dots a_i, a_{i+1}\, C_{p(j+2)} \dots C_{p\bar{p}}\, \gamma)$$

$$\Rightarrow (a_1 \dots a_i a_{i+1}, C_{p(j+2)} \dots C_{p\bar{p}}\, \gamma) .$$

Therefore, $(p, j+1, f)$ is an $(i+1)$-state of a derivation of $a_1 \dots a_{i+1}$. □

Theorem 4.5.2

If RECF$(G, a_1 \dots a_n)$ accepts, then $a_1 \dots a_n \in L(G)$.

Proof

Since $a_1 \dots a_n$ is accepted, $S_n = \{(0, 2, 0)\}$ and $\exists\ q \in F$ such that $Q_n = (q, x_n, P_n)$ for some x_n and some P_n. By Theorem 4.5.1, $(0, 2, 0)$ is an i-state of a derivation of $a_1 \dots a_n$. Hence,

$$\pi_o = (e, D_o) \overset{*}{\Rightarrow} (e, D_o\, \gamma)$$

$$\Rightarrow (e, X \dashv \gamma)$$

$$\overset{*}{\Rightarrow} (a_1 \dots a_n, \gamma).$$

But D_o appears in the right-hand side of no production, so that $\gamma = e$. Therefore

$$(e, D_o) \overset{*}{\Rightarrow} (a_1 \dots a_n, e) .$$

Since the first component of Q_n equals $\delta(q_o, a_1 \dots a_n)$, the latter is an FST accepting state. Hence, $a_1 \dots a_n \in L(G)$. □

The second half of the validity proof essentially consists of showing that Algorithm 4.3.1 can imitate the steps of an ECF generation.

Theorem 4.5.3

If $(p, j, f) \in S_i$ and $(a_1 \dots a_i, C_{p(j+1)}) \overset{m}{\Rightarrow} (a_1 \dots a_\ell, e)$, for some $m > 0$ then $(p, j+1, f) \in S_\ell$.

Proof (By induction on m)

Basis. m = 1. There are two cases:

Case 1. If $C_{p(j+1)} \in \Sigma$, then $\ell = i + 1$ and $C_{p(j+1)} = a_{i+1}$. Hence, $(p, j+1, f)$ is added to $S_{i+1} = S_\ell$ by the scanner.

Case 2. If $C_{p(j+1)} \in I$, then we have $C_{p(j+1)} = D_q$ for some active production with index q; i.e., $(D_q \rightarrow \gamma) \in \mathbb{P}(P_o, T(a_1 \ldots a_i))$. Further,

$$(a_1 \ldots a_i, D_q) \Longrightarrow (a_1 \ldots a_i, \gamma)$$
$$= (a_1 \ldots a_\ell, e).$$

From the equality, it follows that $i = \ell$ and $\gamma = e$. The predictor acting on $(p, j, f) \in S_i$ adds $(q, 0, i)$ to S_i. The completer acting on $(q, 0, i)$ and (p, j, f) adds $(p, j+1, f)$ to S_i. Since $i = \ell$, $(p, j+1, f) \in S_\ell$.

Induction. Suppose the theorem is true for $m \leq k$ and $(p, j, f) \in S_i$, $(a_1 \ldots a_i, C_{p(j+1)}) \overset{k+1}{\Longrightarrow} (a_1 \ldots a_\ell, e)$, with $k \geq 1$. Since $k \geq 1$, $C_{p(j+1)} \in I$. Hence, the derivation may be written

$$(a_1 \ldots a_i, C_{p(j+1)}) = (a_1 \ldots a_i, D_q)$$
$$\Rightarrow (a_1 \ldots a_i, C_{q1} \ldots C_{q\bar{q}})$$
$$\overset{k}{\Longrightarrow} (a_1 \ldots a_i \ldots a_\ell, e),$$

where $(D_q \rightarrow C_{q1} \ldots C_{q\bar{q}}) \in \mathbb{P}(P_o, T(a_1 \ldots a_i))$.

Since $k \geq 1$, it is guaranteed that $C_{q1} \ldots C_{q\bar{q}} \neq e$. Hence, $\exists$ integers $t_0 \leq t_1 \leq \ldots \leq t_{\bar{q}}$ such that $t_0 = i$, $t_{\bar{q}} = \ell$, and $\forall\, r\ (1 \leq r \leq \bar{q})$,

$$(a_1 \ldots a_{t_{r-1}}, C_{qr} \ldots C_{q\bar{q}}) \overset{c_r}{\Longrightarrow} (a_1 \ldots a_{t_r}, C_{q(r+1)} \ldots C_{q\bar{q}}).$$

Since $\sum_{r=1}^{\bar{q}} c_r = k$, $c_r \leq k$ $(1 \leq r \leq \bar{q})$.

Hence, the induction hypothesis is applicable to each of these derivations.

The predictor acting on $(p, j, f) \in S_i$ adds $(q, 0, i)$ to S_i. For each $r = 1, \ldots, \bar{q}$, the following argument applies. Since $(q, r-1, i) \in S_{t_{r-1}}$ and $(a_1 \ldots a_{t_{r-1}}, C_{qr}) \overset{c_r}{\Longrightarrow} (a_1 \ldots a_{t_r}, e)$, it follows that $(q, r, i) \in S_{t_r}$. By induction on r, $(q, \bar{q}, i) \in S_{t_{\bar{q}}} = S_\ell$.

The completer acting on $(q, \bar{q}, i) \in S_\ell$ and $(p, j, f) \in S_i$ adds $(p, j+1, f)$ to S_ℓ. □

Theorem 4.5.4

If $a_1 \ldots a_n \in L(G)$, then RECF$(G, a_1 \ldots a_n)$ accepts.

Proof

If $a_1 \ldots a_n \in L(G)$, then

(1) $(e, D_o) \overset{m}{\Longrightarrow} (a_1 \ldots a_n, e)$ for some $m > 0$,

(2) $\delta(q_o, a_1 \ldots a_n) \in F$.

From the assumed special form of the grammar, (1) is equivalent to

(1′) $(e, X) \overset{\ell}{\Longrightarrow} (a_1 \ldots a_{n-1}, e)$ for some $\ell > 0$.

The initialization step of the algorithm guarantees that $(0, 0, 0) \in S_o$. From (1′) and Theorem 4.5.3, it follows that $(0, 1, 0) \in S_{n-1}$. Since $C_{02} = a_n = \dashv$, the scanner acting on $(0, 1, 0) \in S_{n-1}$ adds $(0, 2, 0)$ to S_n. Since "$\dashv$" appears in no other production, $(0, 2, 0)$ is the only state in S_n. Finally, since $\delta(q_0, a_1 \ldots a_n)$ is the first component of Q_n, RECF$(G, a_1 \ldots a_n)$ accepts. □

4.6 ADDING LOOK-AHEAD

While look-ahead is not strictly necessary, its use may be desirable as a means of gaining efficiency. The case k=1 is particularly attractive as a favorable trade-off point between the savings gained by avoiding incorrect paths and the expense of carrying look-ahead information in the state sets. It was earlier remarked that the look-ahead technique of Earley's algorithm does not carry over to ECF grammars, due to difficulties induced by the variable syntax. In this section, we discuss how these difficulties may be remedied.

The technique we shall discuss has one theoretical shortcoming. Suppose the look-ahead parameter is k. If there are erasing rules, then for certain states corresponding to the erasing rules, the look-ahead will be somewhat less than k. The term "somewhat" will be explicated in the discussion which follows. Here we note that the defect is not really serious. It is rather doubtful that erasing rules will be frequently used in specifying programming languages.

Definition 4.6.1

Let $B_1 \ldots B_\ell \in \underline{V}$ with $\ell \geq k$. We define the function J_k by

$$J_k(B_1 \ldots B_\ell) = B_1 \ldots B_k .$$

For any string, $a_1 \ldots a_i$, of terminals, and any string, $B_1 \ldots B_k$, of k symbols in $\underline{V}$, we define H_k as

$$H_k(a_1 \ldots a_i, B_1 \ldots B_k) =$$

$$\{w \in \Sigma^* \mid |w| = k \text{ and } \exists\ \gamma \text{ such that } (a_1 \ldots a_i, B_1 \ldots B_k) \overset{*}{\Longrightarrow} (a_1 \ldots a_i\, w, \gamma)\}$$

$$\cup\ \{w \in \Sigma^* \mid w = w_1 w_2, |w| = k, \text{ and } (a_1 \ldots a_i, B_1 \ldots B_k) \overset{*}{\Longrightarrow} (a_1 \ldots a_i w_1, e)\}.$$

Remark. The second clause in the definition of H_k provides that if $B_1 \ldots B_k$ generates a string w_1 whose length is less than k, then an acceptable w is obtained by the concatenation of w_1 with any terminal string w_2 such that $|w_2| = k - |w_1|$. This provision is required precisely because one of the Bj's may be erased.

Definition 4.6.2

To provide look-ahead, a state is redefined to be a quadruple (p, j, f, α) where p, j, f are integers and α is a string of k elements of V.

Algorithm 4.6.1 (ECF recognizer with look-ahead)

This is a function, RECFL, of three arguments: an ECF grammar G, a terminal string $a_1 \ldots a_n$, and a look-ahead parameter k. It is computed as follows:

Let $a_{n+j} = \dashv$ $(1 \leq j \leq k)$.

Let S_i be empty $(1 \leq i \leq n)$.

Let P_o be as specified in G.

Let $S_o = \{(0, 0, 0, \dashv^k)\}$.

Let $Q_o = (q_o, e, P_o)$.

Let i = 0 and go to LOOP.

LOOP:

Process the states of S_i in order, performing one of the following on each state $s = (p, j, f, \alpha)$:

1. (Predictor) If $j \neq \bar{p}$ and $C_{p(j+1)} \in I$, then $\forall\ \phi_q \in P_i$ such that $D_q = C_{p(j+1)}$, add $(q, 0, i, J_k(C_{p(j+2)} \ldots C_{p\bar{p}}\alpha))$ to S_i.

2. (Scanner) If $j \neq \bar{p}$ and $C_{p(j+1)} \in \Sigma$, then if $C_{p(j+1)} = a_i$, then add $(p, j+1, f, \alpha)$ to S_{i+1}.

3. (Completer) If $j = \bar{p}$ and if $a_{i+1} \cdots a_{i+k} \in H_k(a_1 \cdots a_i, \alpha)$, then $\forall\ (q, \ell, g, \beta) \in S_f$ such that $C_{q(\ell+1)} = D_p$, add $(q, \ell+1, g, \beta)$ to S_i.

If S_{i+1} is empty, then reject.

If $i = n-1$, $S_n = \{(0, 2, 0, \dashv^k)\}$, and $\exists\ q \in F$ such that $Q_i = (q, x_i, P_i)$, then accept.

Otherwise, let $i = i + 1$, let Q_i be computed as described in Section 4.3, let P_i be its 3^{rd} component, and go to LOOP. end

The above algorithm differs from its predecessor in that for each predicted rule application, $(q, 0, i)$, it carries along the symbol string, γ, which must follow a successful application of the rule. When the right-hand side of the rule has been construed in the input string, the completer verifies that the next k symbols are consistent with γ.

This differs from the look-ahead of Earley's algorithm in two respects. (1) Earley's predicts a terminal string when predicting a rule and carries along this terminal string. Our algorithm defers evaluation of the predicted terminal string until the rule has been successfully applied. (Note that this technique may be profitably employed in the context-free case to reduce the number of states in a state set.) (2) When the syntax rules are fixed, H_k depends only on its second argument and may be computed for all argument values, independently of the recognition process. For ECF grammars, this is not possible. It is necessary to either (a) compute each predicate $a_{i+1} \cdots a_{i+k} \stackrel{?}{\in} H_k(a_1 \cdots a_i, \alpha)$, or (b) compute, for each symbol scanned,

a table of $H_k(a_1 \ldots a_i, \alpha)$ for all possible strings α containing k symbols. For large k, either method becomes prohibitively expensive. However, the case of interest is k = 1, and for that case the computation is reasonable.

The procedure to calculate $H_1(a_1 \ldots a_i, B)$ is as follows:

$$E_i = \{B \mid (a_1 \ldots a_i, B) \overset{*}{\Longrightarrow} (a_1 \ldots a_i, e)\},$$

$$L_i(A) = \{C \mid (A \rightarrow B_1 \ldots B_m\, C\, \gamma) \in P_i \text{ and } B_\ell \in E_i \ \ (1 \leq \ell \leq m)\},$$

$$L_i^*(A) = L_i(A) \cup \{C \mid C \in L_i^*(B) \text{ and } B \in L_i^*(A)\},$$

$$H_1(a_1 \ldots a_i, B) = \begin{cases} \Sigma & \text{if } B \in E_i \\ L_i^*(B) \cap \Sigma & \text{if } B \notin E_i. \end{cases}$$

Instead of performing this calculation for each i, it is possible to compute E_i, L_i, and H_1 by incremental techniques, updating these sets for each input symbol read.

It should be noted that the ECF recognizer with look-ahead does <u>not</u>, in the strict sense, perform k-symbol look-ahead. Consider some state $(p, \bar{p}, f, B_1 \ldots B_k)$ being processed by the completer. If $B_\ell \rightarrow e$ for some ℓ $(1 \leq \ell \leq k)$, then it may be that $(a_1 \ldots a_i, B_1 \ldots B_k) \overset{*}{\Longrightarrow}$ $(a_1 \ldots a_i\, w, e)$ with $|w| < k$. H_k has been "fixed up" to include all terminal strings of length k with initial substring w; hence, the predicate $a_{i+1} \ldots a_{i+k} \overset{?}{\in} H_k(a_1 \ldots a_i, B_1 \ldots B_k)$ will be true. However, for the substring $a_{i+|w|+1} \ldots a_{i+k}$, this test is trivially satisfied. The <u>effective</u> look-ahead is not k, but rather $|w|$.

4.7 PRODUCING A PARSE

It may appear that the discussion of the preceding four sections is very much beside the point: our real interest is in a parse algorithm, not in a recognition device. However, such a charge of irrelevancy would be misplaced. The algorithm, with or without look-ahead, is so constructed that modifying it to produce a parse is a trivial matter: the method is identical to that used by Earley.

For the sake of simplicity, we discuss the recognizer without look-ahead. When the completer acts on states $(p, \bar{p}, f) \in S_i$ and $(q, j, g) \in S_f$ to add $(q, j+1, g)$ to S_i, this may be interpreted as: the symbol $C_{q(j+1)} = D_p$ has been construed in the input string by means of the production $(D_p \rightarrow C_{p1} \ldots C_{p\bar{p}}) \in P_f$. To obtain a parse, this interpretation is recorded by constructing a pointer from $C_{q(j+1)}$, in the production $D_q \rightarrow C_{q1} \ldots C_{q\bar{q}}$, to the production $D_p \rightarrow C_{p1} \ldots C_{p\bar{p}}$. If this sort of action is taken for each step of the completer, a complete parse is obtained. When the algorithm terminates, the state $(0, 2, 0) \in S_n$, and in the production $D_o \rightarrow X \dashv$ pointers lead from the symbol X to its parse.

Since this technique is precisely that of Earley, further elaboration here would be redundant. We refer the reader to Earley's paper for a complete discussion.

4.8 PRACTICAL APPLICABILITY

For an analysis algorithm to be of practical utility for an extensible language or language system, it must satisfy several criteria. One such criterion is generality: the class of languages or

grammars which it handles should be as large as possible and well defined, hopefully in a "natural" fashion. In particular, it should be possible to specify extensions to the syntax without undue worry as to whether such extensions will be acceptable to the analysis algorithm. This trait is, of course, possessed by the above algorithm: it handles any ECF grammar whatever.

Another criterion, applicable to all analysis algorithms, is that the algorithm should be economical in time and space. While the time and space bounds for the recognition algorithm exhibited above are of order n^8, it appears that far better performance will be obtained in practice. One distinct virtue of Earley's algorithm is that in most cases of interest it does far better than its n^3 bound. This also occurs in the ECF case.

It will be recalled that the possibility of a growing production set adds a potential factor of n^5 in parse time. However, this assumes that most of the input string is used to specify new productions. For the expected case, in which the production sets P_i are only small perturbations about P_o, this factor will actually be only somewhat larger than unity.

Further, Earley notes that most LR(k) grammars will parse in time n using his algorithm, even with no look-ahead. If we consider an ECF grammar G and a string $a_1 \ldots a_n \in L(G)$ such that P_i is LR(k) for all i $(0 \leq i \leq n)$, we may expect this result to carry over. Most programming languages – and indeed most natural grammars for programming languages – appear to be LR(k) or LR(k) with only a few exceptions. For example, Korenjak [Kor67] has exhibited an LR(1)

grammar which closely approximates the syntax of Algol 60. Hence, it may be expected that for most ECF programming languages and their terminal strings, the parse time will be of order n. Further, since the production set is variable, it will be possible to confine departures from time-n behavior to local (and hopefully short) sections of the terminal string.

Another desirable trait of a practical analysis algorithm is that it allows error detection and recovery. Irons [Irons63] has observed that most Algol or Fortran programs submitted to a compiler are syntactically incorrect. Hence, pinpointing errors to allow partial automatic correction – or even the production of intelligent error messages – is a problem whose solution is of real significance to the language user. The basic technique used by Irons carries over to the ECF parse algorithm. All possible parses are carried along in a left-to-right scan. An error is detected when no possible parse can be continued (i.e., S_{i+1} is empty), and this is generally very close to the point in the string at which the error occurred. Recovery action to be taken depends on the specific language and language system; hence, its discussion is beyond the scope of this paper. However, it is clear that the information needed to perform recovery is available.

Section 5. NON-FORMAL PROPERTIES

In this section, we deal with a variety of subjects not appropriate or amenable to formal treatment. We showed earlier that the emptiness problem is undecidable for ECF grammars; we now wish to comment on the possible implications of this result. We also explore some meta-topics: an assessment of the formalism with respect to its descriptive power and algebraic properties, and a discussion of generalizations and their properties.

5.1 COMMENTS ON THE UNDECIDABLE EMPTINESS PROBLEM

We first note that the result appears central to the notion of an extensible syntax. The construction used in Theorem 3.4.1 is based on fundamental aspects of the model rather than on accidental features. It requires only that new productions be of unbounded length and that they be obtained from the terminal string by a finite state mapping. The first requirement has been shown necessary if one is to obtain anything beyond the context-free languages (Section 3.1). The second reflects a necessary syntactic freedom in the form of legal strings; it will often be desirable to state syntax extensions in some form other than explicit productions. Hence, it appears that an undecidable emptiness problem may be characteristic of language formalisms which admit an expansion of their syntax.

It is believed by some that this is a grave weakness in a formalism for language description. For example, Fischer [Fisch68] argues that a syntax formalism which has an undecidable emptiness problem fails

to directly describe its languages:

> It seems reasonable to assume that if one cannot even tell from the description of the language whether or not there are any sentences, then that language is not directly described.

We take issue with this argument. On philosophical grounds we find it untenable, for it imposes an overly strong restriction on the notion of "directly describe". The same criterion and argument shows that Algol does not directly describe algorithms, for it is undecidable whether Algol programs halt.

Pragmatically, the argument is also weak, for it guards against a danger that will not occur in practice. If the designer of a programming language using a grammar cannot explicitly exhibit one or more strings generated by the grammar, then there is a "bug" in the language. While it is quite possible to specify ECF grammars whose languages are empty or of unknown emptiness, such grammers are not relevant to the task of language description and will be avoided.

We note that, in general, undecidability results for a class of formal objects are often no obstacle to the use of these objects. For example, an invalid argument could be made to show that context-free grammars fail to directly describe programming languages because the ambiguity problem for context-free grammars is undecidable. This is false precisely because in cases of interest one can insure non-ambiguity by special case arguments.

5.2 RELATION TO CANONIC SYSTEMS

Some insight into the generative power of ECF grammars may be obtained by comparison with other formal systems that have been proposed for the description of programming languages. As we shall show, canonic systems [Donov67], [Led67] invite such comparison. We refer the reader to the cited papers for a full discussion of their theory and application. Here, it suffices to explain that canonic systems are a notational variant of Post's canonical systems, adapted to the specification of programming languages. A canonic system specifies a set by a finite sequence of rules, each of the form:

$$a_1 \ \underline{\text{set } A}_1 \ \& \ \ldots \ \& \ a_n \ \underline{\text{set } A}_n \vdash\!\!- \ b \ \underline{\text{set } B},$$

interpreted as:

if $a_i \in \underline{\text{set } A}_i \quad \forall\ i \quad (1 \leq i \leq n)$,

then it may be asserted that $b \in \underline{\text{set } B}$.

The reader familiar with canonic systems will note two significant traits shared by these systems and ECF grammars.

(1) Information can be stored in sets, allowing coordination of separated segments of the terminal string – a mechanism and facility absent from context-free grammars. Canonic systems perform the storage directly, by set membership. ECF grammars use generated productions, usually I-restricted, to store such data.

(2) Canonic systems, as modified by Ledgard [Led67], are self-extending in the following sense. The form, F, for canonic rules is formally complete but rather austere. It may be usefully extended to a more readable form, F^*, by a number of extensions. Using rules of

form F^*, it is possible to describe conveniently the syntax (and semantics) of a programming language by means of a canonic system C. Now the form F^* can itself be described in terms of F by a canonic system D. If C contains D as a subset, then C is self-descriptive and the formalism is self-extending. This is somewhat reminiscent of Example 2.3.3, in which an initial production set is used to describe the form of legal context-free grammars, and a context-free grammar thus generated is itself the generator of a string.

Having made the above observations of similarity, we raise the question of specific relation. We ask: how do ECF grammars differ from canonic systems? Formally, the answer is simple. Canonic systems generate the recursively enumerable sets; since ECF grammars generate only a subset of the recursive sets, they are strictly less powerful. However, a purely formal exposition is not altogether satisfying. It still may be asked: just what can canonic systems do that an ECF grammar cannot?

In answering the second question, it will be useful to visualize the behavior of an ECF grammar by means of a block diagram:

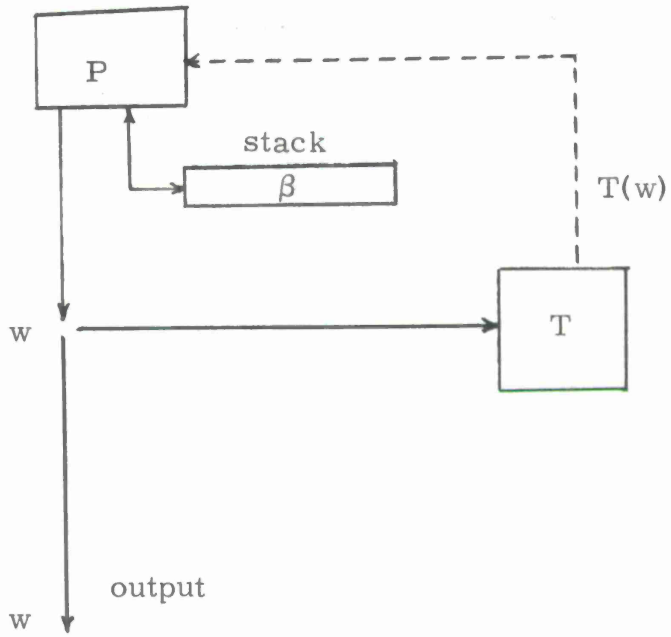


Fig. 5.2.1 Block Diagram of an ECF Grammar

This is to be interpreted as follows. P is the set of local productions. For some instantaneous description, $\pi = (w, \beta)$, represent w as a string already emitted by the generator and β as being on a stack. The finite state transducer T maps w into T(w) and thereby changes P. Solid lines in the diagram represent data flow; dashed lines represent flow of productions.

Consider augmenting the above block diagram to allow modification of the output by a finite state transduction:

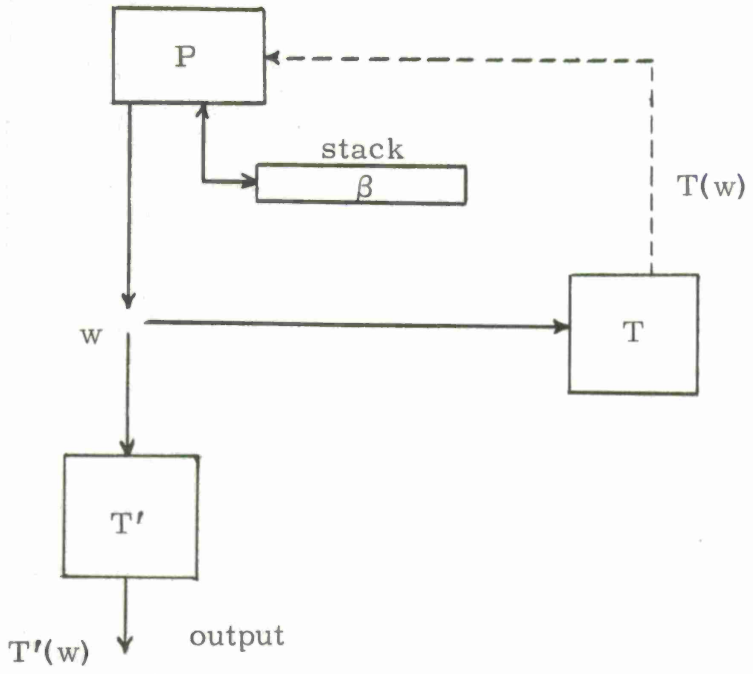


Fig. 5.2.2 Block Diagram of an Augmented ECF Grammar

Since this differs from the previous diagram only by the addition of a finite state transduction, it may appear that the class of grammars this represents does not differ greatly from the ECF. However, this is not the case. It will be recalled that Theorem 3.4.1 demonstrates that for any Turing machine M and initial configuration C, there exists (effectively) an ECF grammar $G = \mathcal{G}(M, C)$ whose strings consist of sequences of Turing machine configurations imitating a legal derivation. An additional finite state transduction (T′), if allowed, can be used to erase all of the string except the initial (or halting) configuration. It follows immediately that grammars whose operation is described by Figure 5.2.2 generate the recursively enumerable sets and hence are equivalent to canonic systems.

Figure 5.2.1 differs from Figure 5.2.2 precisely in that all productions in the former model must be derivable from the output string, whereas in the latter model, erasing by the second FST lifts this restriction. This is the essential difference between canonic systems and ECF grammars: a canonic system may perform an unbounded amount of computation which never appears as explicit output; an ECF grammar cannot.

5.3 ON RESTRICTED CASES

In Section 3.5, we discussed two restrictions on ECF grammars. We demonstrated that these restrictions are non-trivial; i.e., the families of languages so generated are proper subfamilies of the ECF languages. Given this formal result, it may still be asked whether either of these subfamilies would, in practice, be an adequate substitute for the ECF.

For the I-restriction, a negative answer is immediate. Since it restricts new productions to have terminal strings as right-hand sides, it prohibits recursive definition of new syntactic classes. It thereby rules out most of the power of context-free productions. A syntax extensible under such conditions would be of little interest.

The rule number bounded restriction, with some constant k, appears to be somewhat more acceptable. By choosing k large enough, one can be assured that the restriction will rarely be noticed. Further, were the parser of Section 4 implemented on a real computing machine, one could be equally assured that such a bound would be imposed by the implementation. However, this argument must be rejected. When

studying the properties of various automata, it is recognized that: (1) all strings of interest are shorter than some finite constant, say 10^{100}; (2) all realizations of these automata by real computing machines impose restrictions due to a finite address space, say 2^{36}. It would be a mistake, however, to conclude that all automata should be studied as finite state automata. For most practical purposes, 2^{36} is an adequate approximation to infinity and the properties of interest are those obtained by ignoring the limitations imposed by finiteness. It is for this reason that we choose not to use rule number bounded grammars as our formalism: the unbounded case far better embodies the intuitive notion of an extensible syntax.

5.4 SOME COMMENTS ON THE FORMALISM

In the preceding two sections, we have dealt with several possible modifications of the ECF formalism. We wish to continue this discussion, assessing the chosen formalism in comparison to its possible generalizations and rivals.

It should be noted that the family of ECF languages has two prominent characteristics which set it apart from most other generalizations of context-free languages: (1) it has a recognition algorithm which runs in at worst polynomial time; (2) it has very poor algebraic properties (e.g., non-closure under even length-preserving homomorphism. The former, coupled with an expected linear time for common cases, makes it possible to use ECF grammars in the specification of a practical programming language. The latter makes the theoretical study of ECF grammars somewhat difficult. The two characteristics

are, of course, intimately related. A polynomial parse time is possible because generation proceeds left to right (only the leftmost intermediate symbol may be rewritten) and because productions are produced deterministically from the terminal string. However, the strictly left-to-right generation induces asymmetries in the family of ECF languages; for example, it is not closed under reversal. (We have not given a formal proof of this, but the result should be obvious.) Were we to allow a nondeterministic FST with multiple output streams, we could obtain additional algebraic properties, e.g., closure under non-erasing homorphism. This would, however, make a parse far more time-consuming. In short, many features of the formalism which make possible an efficient parse are responsible for the lack of "nice" algebraic properties. In choosing the ECF formalism, we chose to sacrifice the latter to the demands of the former. This choice is appropriate to the purpose at hand: the description of extensible programming languages.

We wish to note one possible generalization which is consistent with this choice. It will be recalled that Example 2.3.3 is an ECF grammar, each of whose strings has the form: the encodement of some context-free grammar G followed by a string in the language L(G). This will <u>not</u> generalize to the case where G is an ECF grammar, for the formalism provides no way to specify and use a variable FST. However, an extension to allow this is not at all difficult to add. An FST can be completely specified by a table consisting of lines, each of the form:

$$s_i \qquad a_i \qquad w_i \qquad s_i'$$

to be interpreted as: when in state s_i if the input symbol is a_i, then output string w_i and go into state s_i'. Such a table line can be directly represented by a string:

$$\vdash\ w_i\ //\ a_i \rightarrow w_i\ //\ s_i'\ \dashv$$

and such a string may be emitted by the FST. If we agree to treat substrings with "$\vdash$", "$\dashv$" brackets as new lines in the FST specification table, we immediately obtain a variable FST. Clearly, the parse algorithm of Section 4 can be modified to perform the necessary actions to imitate this additional variability with, at worst, a polynomial increase in time. As usual, if this variability is used circumspectly, then the increase will in fact be only a small factor. We have refrained from introducing this generalization into the formalism only because no clear-cut application for it could be found. In the absence of any demonstrated need for the facility, simplicity dictates its omission.

Section 6. CONCLUSION

We shall conclude this paper by cleaning up a number of loose ends and displaying explicitly a few others. This will include a survey of research topics for future work and comments on the relation of ECF grammars to the larger topic of extensible programming languages. By and large, these categories are disjoint; the few exceptions merit special consideration.

6.1 OPEN PROBLEMS

The formal theory of ECF languages, as developed in Section 3, contains a few open questions. We do not know, for example, whether the family is closed under inverse homomorphism, or whether it is contained in the context-sensitive languages or the scattered-context languages. Also, we do not have a characterization of the output string emitted by the FST. It is clearly not context-free – but is it, for example, ECF?

In addition to the truly open questions, we have a number of conjectures for which we lack suitable proofs. For example, it would be of interest to prove that the ability to form new intermediate symbols (i.e., members of V_M) is formally required. We show in Appendix I that context-free grammars lose power if the set of non-terminals is bounded, but the proof does not carry over to establish the desired result concerning ECF grammars. Turning to a classical topic in formal language theory, we conjecture – but cannot prove – that there exist inherently ambiguous ECF languages. (Note that this does not follow from the inherent ambiguity of context-free languages.)

The parse algorithm invites a number of interesting, but rather difficult, questions. It would be of interest to prove the conjecture that strings $a_1 \ldots a_n$ such that P_i is LR(0) for all i can be recognized in time n. More generally, a characterization of time-n grammars would be useful. Some theorems relating look-ahead to recognition time would also be useful. There is the practical question: should look-ahead be used in some given language system? This will likely depend on the language, and indeed on the particular string, but some rule-of-thumb for common cases should be possible. We assume that either k=0 or k=1 will be optimal, but it is not clear which. Practical experience in applying the algorithm may be the only way to make a choice.

6.2 APPLICATION TO EXTENSIBLE LANGUAGES

In the Introduction, we delimited the province of this paper to issues in syntactic extension. We now wish to lift this restriction and discuss other aspects of extensible languages which are relevant to a variable syntax.

We assume that the parser produces as output a tree structure which represents a complete parse of the input string. With an appropriate mechanism for data type definition, this tree may be treated as a structured data object (of mode program) and hence is suitable for manipulation by the semantic interpreter. Note that this requires that an appropriate mode declaration be associated with each new production, so that an instance of a new production construed in the input string be interpretable as a data object.

As in the usual formal model for programming languages, the

semantic interpreter operates on the data object program to produce some "meaning". It was noted in the Introduction that along with each new production, $A \rightarrow \alpha$, it is necessary to specify semantics, e.g., the meaning of A in terms of α. We add here that this semantic specification completes the mode definition of the production.

It is possible that a string will be ambiguous with respect to the syntax. Such ambiguities will be represented in the tree structure as multiple parses of some intermediate symbol, and can thereby be identified. We assume that tht interpreter will choose one of the parses, disambiguating on semantic grounds. Making this choice may be a nontrivial problem. Indeed, study of semantic disambiguation in programming languages is a largely unexplored field. Consideration of the topic would be outside the scope of this paper. Here, we merely point out that if the choice is formally specified and hence well defined, this procedure seems perfectly acceptable. It may prove very useful in allowing concise specification of certain language constructs for which an unambiguous syntax would be cumbersome.

We introduced the notion of extensible syntax by a hypothetical extension of Algol, in which productions were declared in blockheads and had their scope determined by block scoping. In the interest of generality, we promptly abandoned this scope rule and replaced it with a formalism in which the scope of a production is essentially the program text which lies between the points at which it is added and deleted. To conform to the well-established tradition of block scoping in programming languages, it might be useful to include block-scoped productions as a special case. This requires only that productions added in

a block be deleted at the block end. This could, of course, be imposed as a requirement on the source text string, just as one could require that declared variables be explicitly "undeclared". It seems preferable, however, to allow the semantic interpreter to handle the matter. We need only allow a limited interaction between the parser and interpreter; in particular, the latter is permitted to delete productions from the local production set. This is not strictly permitted in the framework of the ECF formalism, but in this case the departure is not significant. We could, for example, define a strict language in which explicit deletion of productions (and variables) is required, and then specify text transformations which map the desired language into the strict language.

In conclusion, we wish to note the analogy between the notion of procedures and the notion of ECF grammars. The former allows variable semantics by declaration and subsequent use of program schema. The latter allows variable syntax by means of declaration and use of structural forms. Taken in concert, the two should permit extensible languages with rich, fluent dialects.

APPENDIX I

A THEOREM ON CONTEXT-FREE GRAMMARS

We prove a remark, made in Section 2.1, that the family of languages definable by context-free grammars becomes increasingly large as the number of non-terminal symbols is increased.*

Definition. Let $G = (V, \Sigma, P, X)$ be a context-free grammar. The rank of G is defined to be the number of symbols in $V - \Sigma$.

Theorem

Let Σ be a fixed terminal vocabulary of at least two symbols. Then $\forall\ k \geq 1$, $\exists$ a context-free language $L_k \subseteq \Sigma^*$ such that for all context-free grammars, G', if $L_k = L(G')$ then the rank of G' is $\geq k$.

Proof

With no loss of generality, we may assume that $0, a \in \Sigma$.

Consider the schema of productions:

$$A_1 \rightarrow 0\,a^1\,0\,A_1 \mid A_2$$

$$A_2 \rightarrow 0\,a^2\,0\,A_2 \mid A_3$$

$$\vdots$$

$$A_N \rightarrow 0\,a^N\,0\,A_N \mid 0\,0\,.$$

For each k, consider $G_k = (V_k, \Sigma, P_k, A_1)$ with $V_k = \{A_1, A_2, \ldots A_{2k}\}$ and P_k given by carrying the schema up to $N = 2k$. Then let

$$L_k = L(G_k)$$

$$= \{(0a^1 0)^{i_1} (0a^2 0)^{i_2} \ldots (0a^{2k} 0)^{i_{2k}} \mid i_1, i_2, \ldots i_{2k} \geq 1\}.$$

---

*It was brought to our attention after this work had been completed that a similar result was previously obtained by J. Gruska [Gru67].

Now let $G = (V, \Sigma, P, X)$ be any context-free grammar such that $L(G) = L_k$. We shall show that the rank of G is $\geq k$. The proof uses the following lemma proved by Odgen [Odg68].

Lemma

For every context-free grammar $G = (V, \Sigma, P, X)$, $\exists$ an integer p such that for every string $w \in L(G)$, if p or more distinct positions of w are marked, then $\exists$ $A \in V - \Sigma$ and strings $\alpha, \beta, \gamma, \delta, \mu \in \Sigma^*$ such that

(1) $X \overset{*}{\Longrightarrow} \alpha A \mu \overset{*}{\Longrightarrow} \alpha \beta A \delta \mu \overset{*}{\Longrightarrow} \alpha\beta\gamma\delta\mu = w$,

(2) γ contains at least one marked position,

(3) either α and β both contain marked positions or δ and μ both do,

(4) $\beta\gamma\delta$ contains at most p marked positions.

We refer the reader to the cited reference for a proof of the lemma. We here note only that it is a generalization of a well-known theorem of Bar-Hillel, Perles, and Shamir [Hop69].

Let $\ell = \max(2p-1, 3)$. Consider the string

$$w_1 = (0a^1 0)^{\ell}\ 0a^2 0 \ldots 0a^{2k} 0,$$

and mark the 1^{st}, 3^{rd}, $\ldots$ ℓ^{th} of the a's. Applying the lemma, $\exists$ $A_{i_1} \in V - \Sigma$ and strings $\alpha_1, \beta_1, \gamma_1, \delta_1, \mu_1 \in \Sigma^*$ such that

(1) $X \overset{*}{\Longrightarrow} \alpha_1 A_{i_1} \mu_1 \overset{*}{\Longrightarrow} \alpha_1\beta_1 A_{i_1} \delta_1 \mu_1 \overset{*}{\Longrightarrow} \alpha_1\beta_1\gamma_1\delta_1\mu_1 = w_1$

(2′) either β_1 or δ_1 contains at least one of the marked a's.

We claim that either β_1 or δ_1 has the form "$(0a0)^{k_1}$" for some k_1. The argument is as follows. Suppose β_1 contains a marked "a". From (1), it follows that $\alpha_1\beta_1^n\gamma_1\delta_1^n\mu_1 \in L(G)$ $\forall$ n. From the form of L, this implies that $\beta_1 = (0a^{j_1}0)^{k_1}$ for some j_1, k_1. Since β_1 contains at

least one marked "a", it must be that $j_1 = 1$ and $\beta_1 = (0a0)^{k_1}$ for some k_1. If δ_1 contains a marked "a", the same argument shows that $\delta_1 = (0a0)^{k_1}$, for some k_1. Hence, (2′) may be restated as:

(2) Either β_1 or δ_1 has the form $(0a0)^{k_1}$, for some k_1.

Next, consider the string: $w_2 = 0a0\ (0a^20)^{\ell}\ 0a^30 \ldots 0a^{2k}0$ and let the marked symbols be the 2^{nd}, 6^{th}, 10^{th}, $\ldots$ $(2\ell)^{th}$ instances of the symbol "a". Repeating the argument, it follows that:

(1) $X \overset{*}{\Longrightarrow} \alpha_2 A_{i_2} \mu_2 \overset{*}{\Longrightarrow} \alpha_2\beta_2 A_{i_2} \delta_2 \mu_2 \overset{*}{\Longrightarrow} \alpha_2\beta_2\gamma_2\delta_2\mu_2 = w_2$, for some $A_{i_2} \in V - \Sigma$, and $\alpha_2, \beta_2, \gamma_2, \delta_2, \mu_2 \in \Sigma^*$.

(2) Either β_2 or δ_2 has the form $(0a^20)^{k_2}$, for some k_2.

Repeating the argument 2k times, we have that for all r $(1 \leq r \leq 2k)$, $\exists\ A_{i_r} \in V - \Sigma$, and $\alpha_r, \beta_r, \gamma_r, \delta_r, \mu_r \in \Sigma^*$ such that:

(1) $X \overset{*}{\Longrightarrow} \alpha_r A_{i_r} \mu_r \overset{*}{\Longrightarrow} \alpha_r\beta_r A_{i_r} \delta_r \mu_r \overset{*}{\Longrightarrow} \alpha_r\beta_r\gamma_r\delta_r\mu_r$.

(2) Either β_r or δ_r has the form $(0a^r0)^{k_r}$, for some k_r.

We claim that no three of the A_{i_j}'s may be the same; i.e., $\nexists\ \ell, m, n$ such that:

(1) $1 \leq \ell < m < n \leq 2k$

(2) $A_{i_\ell} = A_{i_m} = A_{i_n}$

We will suppose the contrary and show a contradiction.

As a first step, we claim that if $A_{i_\ell} = A_{i_m}$, then it must be that $\beta_\ell = (0a^\ell0)^{k_\ell}$ and $\delta_m = (0a^m0)^{k_m}$. Indeed, there are only three other cases, each leading to a contradiction. We have that:

$$X \overset{*}{\Longrightarrow} \alpha_\ell\beta_\ell A_{i_\ell} \delta_\ell \mu_\ell \overset{*}{\Longrightarrow} \alpha_\ell\beta_\ell\ \alpha_m\beta_m\ \alpha_\ell\beta_\ell\gamma_\ell\delta_\ell\mu_\ell\ \delta_m\mu_m\ \delta_\ell\mu_\ell$$

Case 1. $\beta_\ell = (0a^\ell 0)^{k_\ell}$ and $\beta_m = (0a^m 0)^{k_m}$. Hence,

$$\alpha_\ell\beta_\ell\alpha_m(0a^m0)^{k_m}\,\alpha_\ell(0a^\ell0)^{k_\ell}\,\gamma_\ell\delta_\ell\mu_\ell\,\delta_m\mu_m\,\delta_\ell\mu_\ell \in L(G)$$

which is impossible, since $\ell < m$.

Case 2. $\delta_\ell = (0a^\ell 0)^{k_\ell}$ and $\beta_m = (0a^m 0)^{k_m}$. Hence,

$$\alpha_\ell\beta_\ell\alpha_m(0a^m0)^{k_m}\,\alpha_\ell\beta_\ell\gamma_\ell(0a^\ell0)^{k_\ell}\,\mu_\ell\,\delta_m\mu_m\,\delta_\ell\mu_\ell \in L(G)$$

which is impossible.

Case 3. $\delta_\ell = (0a^\ell 0)^{k_\ell}$ and $\delta_m = (0a^m 0)^{k_m}$. Hence,

$$\alpha_\ell\beta_\ell\,\alpha_m\beta_m\,\alpha_\ell\beta_\ell\gamma_\ell\delta_\ell\mu_\ell(0a^m0)^{k_m}\,\mu_m(0a^\ell0)^{k_\ell}\,\mu_\ell \in L(G)$$

which is impossible. This proves the first claim.

Now, suppose that $A_{i_\ell} = A_{i_m} = A_{i_n}$, with $1 \leq \ell < m < n \leq 2k$. There are two cases.

Case 1. $\beta_n = (0a^n 0)^{k_n}$. Hence,

$$\begin{aligned} X &\overset{*}{\Longrightarrow} \alpha_n\beta_n A_{i_n}\delta_n\mu_n \\ &\overset{*}{\Longrightarrow} \alpha_n\beta_n\,\alpha_\ell\beta_\ell\gamma_\ell\delta_\ell\mu_\ell\,\delta_n\mu_n \\ &= \alpha_n(0a^n0)^{k_n}\,\alpha_\ell(0a^\ell0)^{k_\ell}\,\gamma_\ell\delta_\ell\mu_\ell\,\delta_n\mu_n \in L(G) \end{aligned}$$

which is impossible.

Case 2. $\delta_n = (0a^n 0)^{k_n}$. Hence,

$$\begin{aligned} X &\overset{*}{\Longrightarrow} \alpha_m\beta_m A_{i_m}\delta_m\mu_m \\ &\overset{*}{\Longrightarrow} \alpha_m\beta_m\,\alpha_n\beta_n\gamma_n\delta_n\mu_n\,\delta_m\mu_m \\ &= \alpha_m\beta_m\,\alpha_n\beta_n\gamma_n(0a^n0)^{k_n}\,\mu_n(0a^m0)^{k_m}\,\mu_m \in L(G) \end{aligned}$$

which is impossible.

Hence, no three of the 2k A's are the same. Therefore, at least k of the A's must be distinct, which proves the theorem. □

APPENDIX II

DEFINITIONS OF SOME STANDARD TYPES OF AUTOMATA

A. Pushdown Store Automata

Definition. A pushdown store automaton (pda) is a 7-tuple = $(K, \Sigma, \Gamma, \delta, \S, q_o, F)$ where K is a finite set of states, Σ and Γ are finite vocabularies, $\S \in \Gamma$, $q_o \in K$ is the initial state, $F \subseteq K$ is the set of final states, and δ is the transition function

$$\delta : K \times (\Sigma \cup \{e\}) \times \Gamma \rightarrow \text{finite subsets of } K \times \Gamma^*.$$

The operation of a pda is described by specifying the form of a machine configuration or instantaneous description and the transitions which take an instantaneous description into its possible successors.

Definition. An instantaneous description (id) of a pda $M = (K, \Sigma, \Gamma, \delta, \S, q_o, F)$ is an element of $(K \times \Sigma^* \times \Gamma^+)$.

The transition between an id_j and a successor id_{j+1} is denoted by $id_j \vdash id_{j+1}$ and is obtained as follows.

(1) $(q, aw, yA) \vdash (q', w, yz)$ if $(q', z) \in \delta(q, a, A)$

(2) $(q, w, yA) \vdash (q', w, yz)$ if $(q', z) \in \delta(q, e, A)$

where $q, q' \in K$, $a \in \Sigma$, $w \in \Sigma^*$, $y \in \Gamma^*$, and $A \in \Gamma$. An id to which neither of the above rules applies has no successor. The ancestral of $\vdash$ is denoted by $\vdash^*$.

Definition. The language accepted by a pda $M = (K, \Sigma, \Gamma, \delta, \S, q_o, F)$ is defined

$L(M) = \{w \in \Sigma^* \mid \exists\ q' \in F \text{ and } u' \in \Gamma^* \text{ such that } (q_o, w, \$) \vdash^* (q', e, u')\}$.

B. Linear-Bounded Automata

Definition. A linear-bounded automaton (lba) is an 8-tuple = $(K, V, \Sigma, \delta, \$, ¢, q_1, F)$, where K is a finite set of states, Σ, V are finite vocabularies, $\Sigma \subseteq V$, $q_1 \in K$ and q_1 is the initial state, $F \subseteq K$ is the set of final states, $\$, ¢ \notin K \cup V$, and δ is a set of quintuples given by

$$\delta_1 \subseteq (K \times V \times K \times V \times \{0, 1, -1\})$$

$$\delta_2 \subseteq (K \times \{¢\} \times K \times \{¢\} \times \{0, 1\})$$

$$\delta_3 \subseteq (K \times \{\$\} \times K \times \{\$\} \times \{-1, 0, 1\})$$

$$\delta = \delta_1 \cup \delta_2 \cup \delta_3.$$

The operation of an lba is described by specifying the form of a machine configuration, or instantaneous description, and the transitions which take an instantaneous description into its possible successors.

Definition. An instantaneous description (id) of an lba $M = (K, V, \Sigma, \delta, \$, ¢, q_1, F)$ is an element of $(\underline{V}^* \times K \times \underline{V}^*)$, where $\underline{V} = V \cup \{\$, ¢\}$.

The transition between id_j and a successor id_{j+1} is denoted by $id_j \vdash id_{j+1}$ and is obtained as follows:

(1) if $(q, Y, q', Z, 0) \in \delta$, then $(w_1, q, Y w_2) \vdash (w_1, q', Z w_2)$

(2) if $(q, Y, q', Z, 1) \in \delta$, then $(w_1, q, Y w_2) \vdash (w_1 Z, q', w_2)$

(3) if $(q, Y, q', Z, -1) \in \delta$, then $(w_1 X, q, Y w_2) \vdash (w_1, q', XZ w_2)$,

where $w_1, w_2 \in \underline{V}^*, X, Y, Z \in \underline{V}$, and $q, q' \in K$. An id to which none of the above rules applies has no successor. The ancestral of $\vdash$ is

denoted by $\vdash^*$. The language, L(M), accepted by an lba M is defined: $L(M) = \{w \in \Sigma^* \mid$ there exists $w' \in V^*$ and $q \in F$ such that $(e, q_1, ¢w\$) \vdash^* (¢w'\$, q, e)\}$.

C. Turing Machines

Definition. A Turing machine is a 6-tuple = $(K, \Sigma, \Gamma, \delta, q_o, F)$ where K is a finite set of states, Σ and Γ are finite vocabularies, $\Sigma \subseteq \Gamma$, $q_o \in K$ is the initial state, $F \subseteq K$ is the set of final states, and δ is a finite set of quintuples

$$\delta \subseteq (K - F \times \Gamma \times K \times \Gamma \times \{0, 1, -1\}).$$

The operation of a Turing machine is described by specifying the form of a machine configuration or instantaneous description, and the transitions which take an instantaneous description into its possible successors.

Definition. An instantaneous description (id) of a Turing machine $M = (K, \Sigma, \Gamma, \delta, q_o, F)$ is an element of $(\Gamma^* \times K \times \Gamma^+)$.

The transition between id_j and a successor id_{j+1} is denoted by $id_j \vdash id_{j+1}$ and is obtained as follows:

(1) if $(q, Y, q', Z, 0) \in \delta$ then $(w_1, q, Y w_2) \vdash (w_1, q', Z w_2)$

(2) if $(q, Y, q', Z, 1) \in \delta$ then $(w_1, q, YX w_2) \vdash (w_1 Z, q', X w_2)$

(3) if $(q, Y, q', Z, 1) \in \delta$ then $(w_1, q, Y) \vdash (w_1 Z, q', ß)$

(4) if $(q, Y, q', Z, -1) \in \delta$ then $(w_1 X, q, Y w_2) \vdash (w_1, q', XZ w_2)$

(5) if $(q, Y, q', Z, -1) \in \delta$ then $(e, q, Y w_2) \vdash (e, q', ß Z w_2)$

where $w_1, w_2 \in \Gamma^*$, $X, Y, Z \in \Gamma$, $q, q' \in K$, and $ß \in \Gamma$ is a special symbol

which denotes a "blank tape square". An id to which none of the above rules applies has no successor. The ancestral of $\vdash$ is denoted by $\overset{*}{\vdash}$.

Definition. The language accepted by a Turing machine $M = (K, \Sigma, \Gamma, \delta, q_o, F)$ is defined

$$L(M) = \{w \in \Sigma^* \mid \exists\ w', w'' \in \Gamma^*\ \ q_f \in F \text{ such that } (e, q_o, w) \overset{*}{\vdash} (w', q_f, w'')\}.$$

REFERENCES

[Aho68] Aho, A. V. "Indexed Grammars – An Extension of Context-Free Grammars," Journal of the ACM, Vol. 15, No. 4, pp. 647-671, October 1968.

[Aho69a] Aho, A. V. "Nested Stack Automata," Journal of the ACM, Vol. 16, No. 3, pp. 383-406, July 1969.

[Aho69b] Aho, A. V. Private communication.

[Bell68] Bell, J. R. The Design of a Minimal Expandable Computer Language, Doctoral dissertation, Stanford University, December 1968.

[Chea66] Cheatham, T. E. "The Introduction of Definitional Facilities into Higher Level Programming Languages," Proceedings of the AFIPS Fall Joint Computer Conference, Vol. 29, pp. 623-637, November 1966.

[Chea68] Cheatham, T. E. "On the Basis for ELF – An Extensible Language Facility," Proceedings of the AFIPS Fall Joint Computer Conference, Vol. 33, pp. 937-948, November 1968.

[DiFor63] Di Fornio, A. C. "Some Remarks on the Syntax of Symbolic Programming Languages," Communications of the ACM, Vol. 6, No. 8, pp. 456-460, August 1963.

[Donov67] Donovan, J. J., and Ledgard, H. F. "A Formal System for the Specification of the Syntax and Translation of Computer Languages," Proceedings of the AFIPS Fall Joint Computer Conference, Vol. 31, pp. 553-569, November 1967.

[Earl68] Earley, J. An Efficient Context-Free Parsing Algorithm, Doctoral dissertation, Carnegie-Mellon University, August 1968.

[Fisch68] Fischer, M. J. "Grammars with Macro-Like Productions," Mathematical Linguistics and Automatic Translation, Report No. NSF-22, May 1968.

[Gall67] Galler, B. A., and Perlis, A. J., "A Proposal for Definitions in Algol," Communications of the ACM, Vol 10, No. 4, pp. 204-219, April 1967.

[Gar68] Garwick, J. V. "GPL, a Truly General Purpose Language," Communications of the ACM, Vol. 11, No. 9, pp. 634-638, September 1968.

[Gins68a] Ginsburg, S., and Greibach, S. "Abstract Families of Languages," System Development Corporation Report, TM-738/044/00, March 1968.

[Gins67] Ginsburg, S., Greibach, S. A., and Harrison, M. A. "Stack Automata and Compiling," Journal of the ACM, Vol. 14, No. 1, pp. 172-201, January 1967.

[Gins68b] Ginsburg, A., and Spanier, E. H. "Control Sets on Grammars," Mathematical Systems Theory, Vol. 2, No. 2, pp. 159-177.

[Greib68] Greibach, S., and Hopcraft, J. "Scattered Context Grammars," System Development Corporation Report, TM-738/043/00, February 1968.

[Gru67] Gruska, Jozef. "On a Classification of Context-Free Languages," Kybernetika 3 (1967), pp. 22-29.

[Hop69] Hopcroft, J. E., and Ullman, J. D. Formal Languages and Their Relation to Automata, Addison-Wesley Publishing Company, Reading, Massachusetts, 1969.

[Irons63] Irons, E. T. "An Error-Correcting Parse Algorithm," Communications of the ACM, Vol. 6, No. 11, pp. 669-673, November 1963.

[Kas67] Kasami, T., and Torii, K. "Some Results on Syntactic Analysis of Context-Free Languages," Record of Technical Group on Automata Theory of Institute of Electronic Communication Engineers, Japan, January 1967.

[Knu65] Knuth, D. E. "On the Translation of Languages from Left to Right," Information and Control, Vol. 8, pp. 607-639, 1965.

[Kor67] Korenjak, A. J. A Practical Approach to the Construction of Deterministic Language Processors, unpublished paper: RCA Laboratories, Princeton, New Jersey, September 1967.

[Led67] Ledgard, H. F. Canonic Systems: A Self-Extending Formalism for Defining the Syntax and Translation of Computer Languages, unpublished paper: Department of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, Massachusetts.

[Naur63] Naur, P. "Revised Report on the Algorithmic Language Algol 60," Communications of the ACM, Vol. 6, No. 1, pp. 1-17, January 1963.

[Odg68] Odgen, W. "A Helpful Result for Proving Inherent Ambiguity," Mathematical Systems Theory, Vol. 2, No. 3, pp. 191-194 September 1968.

[Ros69] Rosenkrantz, D. J. "Programmed Grammars and Classes of Formal Languages," Journal of the ACM, Vol. 16, No. 1, pp. 107-131, January 1969.

[Whit68a] Whitney, G. E. "A Table Directed Grammar for the Specification of Context-Sensitive Languages," Proceedings of the Second Annual Princeton Conference on Information Sciences and Systems, 1968.

[Whit68b] Whitney, G. E. "The Generation and Recognition Properties of Table Languages," Proceedings of the IFIP Congress 68, Edinburg, pp. B18-B22, August 1968.

[Whit68c] Whitney, G. E. "The Position of Table Languages Within the Hierarchy of Nondeterministic On-Line Tape-Bounded Turing Machine Languages," IEEE Conference Record Ninth Annual Symposium on Switching and Automata Theory, pp. 120-130, October 1968.

[Whit69] Whitney, G. E. "An Extended BNF for Specifying the Syntax of Declarations," Proceedings of the AFIPS Spring Joint Computer Conference, Vol. 34, May 1969.

Chapter 3

THE DESIGN AND FORMAL SPECIFICATION OF EL1

Section 1. INTRODUCTION

In this chapter, we deal with three topics: (1) the design of the base for an extensible programming language, (2) the specification of semantics of programming languages, (3) the application of (2) to (1).

This study therefore draws upon two areas of current research in programming languages: the analysis and modeling of semantics and the quest for extensibility. While there has been an abundance of work in each of these areas, little attention has been given to their interaction. Our interest is motivated less by an aesthetic desire for syncretism than the belief that semantic modeling and language extensibility are necessarily complementary. Neither study is likely to bear fruit alone, whereas taken together they provide a handle on the synthesis of tractible programming languages.

The formal study of semantics predates the study of extensibility by a number of years. The former was given its initial impetus by the Algol report [Naur60] of 1960. The precision and clarity of its BNF syntax specification presented a sharp contrast to the loose and often obscure English language specification of semantics [Dijk62]. True, the syntax was found to be ambiguous [Cant62], but the exactness of the formal description made it possible to localize the difficulty, discuss it unambiguously, and repair it in the revised report [Naur63]. Further, the formal syntax description served as a departure point for a significant

body of research in parsing techniques, formal language theory, and related areas.

The semantic specification, on the other hand, has never been altogether satisfactory. It is far from readable and serves as a reference only with the aid of considerable exegesis. Despite its attempt at precision, it contains a large number of ambiguities many of which are sufficiently complex that they could not be resolved in the revised report (c.f. [Knu67] for a discussion of these).

The disparity between syntactic and semantic specifications invited research into the techniques and formal models which could be brought to bear on the latter. This work is surveyed in section 2.1; here, two points should be noted. (1) A wide variety of approaches were tried based on models ranging from translator writing systems to the λ-calculus. (2) This work was never altogether successful. The desired semantic specifications were often attained via unacceptable circumlocations or at the price of unwieldy bulk. Rarely is a point better explicated in the model than in equivalent English text. Never does such semantic description approach the clarity and utility of syntax description via BNF.

In retrospect, it is apparent that these efforts were predestined to be unsuccessful. The semantic domains to be described were complex, inhomogenous, and generally ad hoc; the injection of these characteristics into the models describing them was inevitable. An analogy with syntax may be useful: while it is in principle possible to devise a formal schema and use it to specify the syntax of Fortran, such efforts would be misplaced. The weakness of attempts at modeling the semantics of many programming languages is often less an indictment of the modeling technique than of the language.

Turning from semantic modeling of languages to their use by the programming community, one finds quite another indictment of most languages. Despite their complexity, they are never sufficiently complete to express all algorithms easily and efficiently. Successive generations are larger, more complex, more expensive and still incomplete. Following the example of assembly languages, it was proposed† around 1965 that development should be shifted from ever larger monoliths to languages containing definition and extension facilities. Supplied with such a language, a programmer would be able to create for himself a dialect appropriate to his needs.

This attempt to aid the language user had an unexpected impact upon the theoretical study of languages. It was soon realized that a programming language capable of extension could be considerably simpler than conventional programming languages: the accretions of special facilities introduced to satisfy various user demands could be removed with the knowledge that they could be obtained, when required, as extensions. The turn to simpler yet more powerful languages made the job of semantic specification simultaneously more practicable and more important. The semantic specification need encompass only the language core, for the semantics of extensions can be derived by projection onto this core. Hence, the specification need cover a considerably smaller domain. On the other hand, this semantic projection brings the core under sharper scrutiny. Since any ambiguity in the core would be propagated throughout, ambiguity must be debarred at the outset. In fact, a stronger condition is required:

---

†The idea seems to have been arrived at independently by a number of researchers. These include Garwick, Ingerman, Lucas, Steel [Gar67], Galler, Perlis [Gal67], Cheatham [Chea66], and Leavenworth [Leav66].

the description must be sufficiently clear that the unique meaning is apparent to the programmer, without the intermediary of a language priesthood.

In the above paragraphs, our description of extensible languages and their extension mechanisms has been deliberately loose. We shall give a more precise description, delineating two classes of extension. In each of these, a formal semantic specification plays an important role.

The analogy with assembly language (i.e., their macro facilities) suggests extension mechanisms designed to permit paraphrase. Given a concept which can, in principle, be expressed in some language, a paraphrase extension is aimed at expressing this concept in a fluent notation or efficient fashion. Following the dictum of Perlis, this notation is chosen to suppress the constant and display the variable. The goal of this work is to allow the programmer to express precisely the meaning of an algorithm, not some equivalent but clumsy and unreadable circumlocation of this meaning.

Since the paraphrase extension facility maps all meaning into the semantics of the language base, this imposes strong requirements on that base. Clearly, its concepts must span a large, interesting space. Of equal importance is that its representations, in particular its representation of data†, be as efficient as possible. A formal semantic specification plays three roles in the design of such a base. (1) It serves as a frame of

---

†For example, linked lists of elements with dynamic type are a perfectly general data representation from which any structure or behavior can be constructed. Their exclusive use would, however, be intolerable in many problem areas. Here as elsewhere in semantic modeling, it is necessary to distinguish between representation and effective representation.

discourse, allowing analysis and comparison of various proposed components of the base. (2) It exposes omissions and inhomogeneities, for the treatment of analogous structures in dissimilar fashion is readily apparent. (3) It brings into the domain of discourse choices which might otherwise be taken for granted and never be subjected to critical examination.

The second class of language extensions is motivated by the observation that certain desired notions may well fall outside the semantic space spanned by the base language. Given the growth of programming and its theory, this seems inevitable, regardless of how well-chosen the base may be. While this limitation is by no means an excuse for slipshod initial design, it must be recognized and dealt with. It can indeed be handled if the notion of extensible language is taken in its broadest sense. A properly designed formalism for semantic specification can, as noted above, describe not only the actual language chosen but also those choices rejected, in fact a wide class of languages. Indeed, the ability to do so is one criterion for assessing the power and generality of a formalism. It may be that while the desired extension cannot be expressed as a paraphrase, it can be defined by means of the semantic model. If the language includes handles on its underlying semantic specification, such a definition can be incorporated into the language. We will refer to such an addition as a metaphrase extension or, more briefly, as a metaphrase.

There will be a number of types of metaphrase extensions. Frequently, the desired extension will be intimately involved in the existing language. For example, a small change to the evaluation rules, perhaps using an existing mechanism in a different fashion will produce a large change in the language. Alternatively, a metaphrase may specify some new domain of discourse, e.g. the addition of pattern-matching facilities to a language which formerly had none. In such cases the metaphrase may act largely

as an independent module with relatively little linkage to existing routines.

One point should be noted. While a metaphrase extension will completely specify the semantics of an extension, there remains the problem of implementing the extension. Unless the semantic model is used as a direct implementation, the metaphrase written in the metalanguage must be translated to a form compatible with the implementation. How this translation is performed depends on both the model and particular implementation. Clearly, its work is facilitated if the model and implementation obtain their results by analogous processes. This implies the use of homologous information structures and evaluation rules so that the model becomes an implementation guide as well as abstract definition.

In this chapter, we develop a base for an extensible language and a technique for its formal specification. Our thesis is that the two activities are necessarily complementary. The formal specification should not be an after-the-fact description of the base but rather serve as a tool to be used in its design. Conversely, while the base must span a large semantic space, it should be kept small and homogenous so as to make practical the task of semantic description. Further, both base and its formal specification are to be so designed as to permit extension, both paraphrase and metaphrase.

Section 2. SURVEY OF PREVIOUS WORK

There already exists a large body of research into both formal semantic specification and extensible languages. In no small measure, our work builds on previous research which in turn builds on earlier work. Hence, a survey of the field is required, to an extent not frequently encountered in the young study of computer science.

A complete survey, doing justice to all the relevant research, would be far beyond the scope of this paper. We have neglected many peripheral issues, for the treatment of which we refer the reader to survey papers by Feldman [Feld68] and Wegner [Wegn69]. Further, we have restricted attention to those papers which are most significant; where equivalent work was carried out in several projects, we have chosen one representative instance. Even so, this section has grown to embarrassing proportions. While believing it necessary, we regret the inclusion of a long survey in this paper and beg the reader's indulgence.

2.1 SEMANTIC SPECIFICATION OF PROGRAMMING LANGUAGES

In the introduction, it was noted that a wide variety of models have been proposed for semantic specification. In contrast to syntactic specification where a satisfactory technique, BNF, was invented almost as soon as a need was recognized, no completely satisfactory semantic technique has yet emerged. The problem, well-recognized but unsolved for several years, has proved a spur to repeated efforts employing many diverse models and sundry variations. Consequently, the field has seen considerable experimentation — most of it valuable.

The next four sub-sections survey and assess the most significant models, analyzing the potentialities and limitations of the various approaches

they take. These include compiler-based models, λ-calculus models, several interpreter models, and the ULD model developed by IBM Vienna Laboratories. Following the survey, we draw some conclusions concerning semantic specification and its relation to programming languages.

2.1.1 Compiler Models

As most programming languages are implemented by means of a compiler, it is attractive to obtain a semantic specification by simply formalizing the process of compilation. Perhaps the most articulate presentation of this position is due to J. Garwick. He proposes, [Gar66], that the semantics of a programming language be defined by a standard compiler for that language. The compiler is to be written in and produce object code for some standard "machine-independent language" suitable for simulation on any normal computer. The meaning of any program is defined to be the outcome of the simulator acting on the output of the standard compiler. The standard compiler would thus serve as an unambiguous definition, guaranteeing the existence of an effective procedure for obtaining the meaning of any program. While the standard compiler would be made simple at the price of inefficiency, it could serve as a standard for the development of better compilers. A new compiler could be certified by running on it all the "fancy cases" [sic] much in the way an algorithm is certified.

The key issue here is the perspicuity of the standard compiler. Clearly, a precise definition of a language can be obtained by anointing at random a compiler for that language; precision is, however, not the only concern. For Garwick's proposal to be non-trivial, the standard compiler must serve as documentation as well as canonical implementation. That is, it would have to be so transparent that one could read it and understand its

operation without resort to a computing machine. While Garwick suggests that this could be achieved by sacrificing efficiency, he presents no example to support the contention. Indeed, it seems unlikely that examples can be found. The difficulty of communicating an algorithm in machine language was one of the motivations for high level languages. As a compiler is generally several orders of magnitude more complex than a single algorithm, machine language would hardly appear to serve as an appropriate vehicle for its communication.

Having appealed to the utility of high level languages, we are invited to investigate whether their substitution for machine language would make Garwick's proposal tractible. We consider one of several efforts made along these lines – FSL [Feld66] – choosing it over others because it was designed specifically as a formal semantic language.†

The underlying compiler model used by FSL is a standard on-the-fly code generation scheme (c.f. [Chea67] for a complete discussion of such techniques). Syntactic analysis is performed by Floyd-Evans productions (c.f. [Chea67]) which are usually used to produce a canonical parse, but which can in principle allow greater generality. Each reduction may optionally call upon a semantic routine written in FSL; the actions carried out by these routines define the meaning of the program. Semantic routines can generate code and/or change the state of the compiler.

Code generation is performed by calls upon abstract code operators, e.g., JUMP, PLUS, MULTIPLY, ASSIGN. The actual generation of code corresponding to the abstract operators is performed by system-defined,

---

†Subsequent to theoretical design, it was implemented in a translator writing system, and later expanded in VITAL [Mond67] which was used for the implementation of LEAP [Rovn68] as well as Algol 60.

machine-dependent code generation routines whose operation is below the level of discourse in FSL. Hence, the problem of code generation is neatly side-stepped in the formal semantic specification. Since code generation is generally messy but semantically tractable, it may be properly regarded as a subsidiary issue; its removal from the semantic model is a useful orthogonalization. Further, by using abstract code operators, FSL avoids commitment to a particular machine, or even a particular machine organization. Hence, the formal definition specifies semantics precisely without making commitments which might result in pragmatic inefficiency.

Changes to the state of the compiler include testing and updating tables, manipulating stacks, and operating on compile-time variables. FSL is itself a fairly complete programming language and contains Booleans and Boolean operations, conditionals, assignments, and procedure calls. In addition, there are a number of builtin functions which abstract situations occurring frequently in compiling; for example, a floating address notation is provided to handle forward reference such as jumps to program locations not yet determined.

One objection to this approach is based on the complexity of the formal semantic language. Its purely algorithmic capabilities are not much weaker than Algol 60; as these are augmented by special builtin functions, the complete language FSL is more in need of semantic explication than most of the languages it has been used to define. Such objections could, in principle, be met by defining the FSL language in FSL (c.f. §4.2). Alternatively, a somewhat simpler semantic language could be used with little loss of descriptive power, provided that a careful choice is made. For example, Wirth and Weber give a formal definition of EULER [Wir66] in which the semantic actions are specified in an "elementary notation for algorithms", a language far simpler than FSL.

A more serious objection applies to compiler models for semantics as a genre. Wegner [Wegn69] has observed that compiler based models rely on the supposedly understood semantics of the target machine; hence, they are "analogous to the solution of a mathematical problem by reducing it to a second problem with known solution." The difficulty with this approach, Wegner observes, is that it often fails to directly explicate the essential nature of the problem being solved. Specifically, the trouble with compiler models is that semantics gets distributed in this two-stage process so that the one-to-one correspondence between structure and meaning is lost. To understand the semantics of a language construct, it is necessary to understand not only the semantic actions which specify code for that construct but also the environment in which that code will run. As this environment does not exist at the time the semantic actions are taken, it must be mentally preconstructed from an understanding of the other actions taken by the compiler and an understanding of the behavior of the program. Further, some of the relevant compiler actions do not occur until after the processing of the construct in question. This diffusion of semantics runs precisely counter to the goals of understanding, analysis, and communication. To be acceptable, a semantic formalism must orthogonalize, not commingle meaning. Hence, despite their initial attractiveness, the compiler models prove somewhat unsatisfactory.

The efforts at producing a more direct explication are divided into two camps, depending on whether or not the λ-calculus is taken to be the canonical form for exegesis. In either camp, most models use an interpreter of some sort.

2.1.2 λ-Calculus Models

In a series of overlapping papers, [Land64], [Land65], [Land66a], and [Land66c], P. Landin has explored the application of the λ-calculus to the analysis and explication of programming language semantics. This work comprises a number of distinct but complementary themes:

(1) demonstrating how certain constructs in programming languages (e.g., auxiliary definitions, parameter bindings, recursive function definitions) can be modeled in the λ-calculus,

(2) specifying the evaluation of λ-expressions by a mechanical procedure which operates by state transitions in the spirit of automata,

(3) demonstrating how the λ-calculus can be syntactically enriched so that it has the appearance of a simple programming language (named "AE"), and semantically augmented to obtain a more general programming language (named "IAE"),

(4) specifying a formal definition of Algol 60 semantics by a function which maps Algol 60 into IAE.

As an illustration of the modeling technique, consider the following fragment in some hypothetical programming language

```
let u = 2p + q;
and v = p - 2q;
and f(x) = sin(5x^2 + 3x);
f(u) + f(v);
```

This may be translated into the equivalent λ-expression

$$[\lambda(u, v, f) . f(u) + f(v)]\ (2p+q,\ p-2q,\ \lambda(x) . \sin(5x^2+3x))$$

In general, mappings of this sort can be specified by a set of transformation rules; the two employed in this translation are

(1) and ⟨variable⟩$_1$ (⟨variable⟩$_2$) = ⟨expression⟩

$\longrightarrow$

and ⟨variable⟩$_1$ = λ(⟨variable⟩$_2$) . ⟨expression⟩

(2) let ⟨variable⟩$_1$ = ⟨expression⟩$_1$;
{and ⟨variable⟩$_i$ = ⟨expression⟩$_i$;}* ⟨expression⟩$_b$;

$\longrightarrow$

[λ(⟨variable⟩$_1$ { , ⟨variable⟩$_i$}*) . ⟨expression⟩$_b$]
(⟨expression⟩$_1$ { , ⟨expression⟩$_i$}*)

Other forms, notably recursive procedure definitions can be modeled in similar fashion.†

The mechanical evaluation of λ-expressions is of little interest to the present discussion except in that it illustrates the technique of defining a language by means of its interpreter. However, what is of interest is topic 4: Landin's semantic specification of Algol 60. Since the λ-calculus is strictly applicative, it models only with great difficulty certain imperative features of programming languages, notably jumps and assignments. To handle these features, Landin adds corresponding primitives to the λ-calculus: program points, and assigners. The resulting language he terms "IAE" (imperative applicative expressions). The formal definition of IAE was to have been specified by a mechanical evaluator related to the evaluator for the pure λ-calculus. However, to the best of our knowledge, this evaluator was never written. This undercuts the model of Algol 60, since the only semantic specification of IAE is a very short English language discussion.

---

† It should be pointed out that the transformation rules are ours, not Landin's; his papers are expository and present the techniques by examples, not formal rules. It should also be noted that the rules, taken to transform in the reverse direction, illustrate the technique of syntactically enriching the λ-calculus so as to mimic the appearance of conventional programming languages.

Landin does define with some rigor an abstract form of Algol 60 and a function which maps abstract Algol into IAE. The treatment of the Algol ⟨for statement⟩ illustrates the general technique as well as a trait typical of Landin's approach: whenever possible, imperative constructs are recast into an applicative rendering. In abstract Algol, a forstatement is defined by

> a forstatement has
>
> a control which is a variable
>
> and a forlist which is a nonnull forlistelement-list
>
> and a body which is a labeled statement

where "variable", "forlistelement-list", and "statement" are similarly defined. The mapping rule for an abstract forstatement is given by a function written in the language AE, which it will be recalled is a paraphrase of the λ-calculus:

```
nforstatementNS =
let D_o, D, X = nlabeled(nstatementN'I')(bodyS)
(D_o,
parallel ( ),
combinelist('for',
        (nlhsN(controlS);
        combinelist('concatenate*',
                map(nforlistelementN)(forlistS)),
        arrangeaspseudoblock(D, X)))
```

where "bodyS", "controlS", and "forlistS" refer to the parts of the forstatement and the various functions such as "parallel", "combinelist", "map", and "arrangeaspseudoblock" are specified by similar definitions.

This mapping rule compiles a call on the IAE function for* with three arguments:

(1) the controlled variable,

(2) a special function, called a stream, which steps through the forlist,

(3) the body to be executed on each iteration.

Finally, for* is a function defined in IAE as follows:

recursive for*(v, S, B) = if ¬ null S( ) then
[v := hS( ); B ; for*(v, tS( ), B)]

This definition should be moderately clear once it is explained that S is a null-adic stream function which models an Algol ⟨for list⟩ in the following sense. When S is called, it produces either NIL or a list of two elements: (1) the first ⟨for element⟩ specified in the Algol program, (2) a function which models, in the same sense as does S, the rest of the ⟨for list⟩. The function for* tests to see if there is a next ⟨for element⟩; if so, it sets v to the next ⟨for element⟩, executes the body B, and calls itself recursively with a modified procedure for producing a ⟨for list⟩.

In comparing this definitional technique to the compiler-based models, several points should be noted. The mapping (compiler if you will) from abstract Algol to IAE is expressed by a purely applicative function. Hence, there are no side effects and the mapping process can be treated statically. The proponents of applicative programming argue that this greatly simplifies the mapping and makes it semantically acceptable. Further, since AE is rigorously equivalent to the λ-calculus, the semantic metalanguage for the mapping phase is well-defined. While it is true that AE is well-defined, it is not at all clear that its use actually simplifies the mapping. If anything, it illustrates one difficulty with applicative programming: notions which are

intuitively expressed by imperatives are tortuously twisted into applicatives. The same criticism applies to the IAE code which is generated by the mapping. Whereas machine code, abstract or otherwise, generated by the compiler-based models is fairly clear, the IAE rendering of familiar Algol forms is often highly counter-intuitive. The stream function used above is typical. Other examples include the Algol conditional expression

if p then a else b

which is represented by the IAE form

if(p) (λ( ) . a, λ( ) . b) ( )

Here, if is a function-producing function defined

if(true) = head

if(false) = head tail

where head and tail are the usual list operators. Circumlocutions of this form are the rule rather than the exception in Landin's work, suggesting that IAE is not really satisfactory as a target language for the explication of Algol or similar programming languages.

Following Landin, intellectually as well as chronologically, there have been a number of other studies using the λ-calculus to explicate features of programming languages, e.g., [Mor68]. However, this work contains no significant advances; it appears the λ-calculus approach has run into a dead end.

Turning, however, from explication to synthesis, it is found that the λ-calculus is a reasonable basis for a programming language. That is, one can take the λ-calculus as a starting point and build languages around it. Since these languages tend to be dominated by their core, they are often elegant and semantically tractible.

Lisp [McCar60] was the first programming language to take this path.† In so doing, Lisp made two contributions to programming languages.

(1) Its evaluation rules are particularly simple and uniform. With only a few exceptions, Lisp stands as a model for its lack of "funny" situations and special cases.

(2) Because of this uniformity, it was possible to give a precise specification of the evaluation process. That this specification is written in Lisp makes it particularly elegant, but this elegance is of secondary interest. The existence of a precise, lucid semantic specification demonstrated that this goal was attainable and inspired attempts at duplication of this precision and lucidity in the specification of other languages.

The work on λ-calculus models may be summarized as follows.

(1) The λ-calculus is a viable semantic tool for the applicative aspects of programming languages. Almost all languages contain some applicative facets such as parameter binding, scope rules, function definition, function-producing functions, and the like. The semantics of these constructs can often be nicely analyzed and explicated in terms of the λ-calculus.

(2) Further, it is possible to design languages whose applicative facets are based directly on the λ-calculus. Such languages can be particularly tractible.

(3) The λ-calculus can be used to model some imperative aspects of programming by recasting imperative notions into applicative ones.†† Less

---

†Subsequently, several other languages were so designed, notably ISWIM [Land66b] and its descendent PAL [Evans68].

††Indeed, since the λ-calculus is effectively equivalent to a universal Turing machine, it is not difficult to show that any imperative notion can be recast into some weakly equivalent applicative notion.

mutable imperative aspects can be handled by a formalism in which the λ-calculus is augmented by the ad hoc addition of a few imperative features. (4) However, the models which result from (3) are at best questionable and often useless.

2.1.3 Interpreter Models

Granting the conclusions of section 2.1.2, a number of researchers have sought other schema whose primitives (or axioms, if you will) more properly reflect the behavior of computers and programming languages. Notably, these include assignment. While this carries the formal theory outside the province of classical mathematics, work by J. McCarthy and his students has shown that such theories can be tractible. In particular, an axiom set including assignment has been shown, [Kapl68], complete and consistent.

Other than agreement on the need for assignment and hence explicit sequencing, the resulting models bear little resemblance to one another. We shall consider three: Van Wijngaarden's, McCarthy's, and ULD.

The formal model of Van Wijngaarden was outlined in two papers [VanW63] and [VanW66] and applied to the formal definition of Algol 60 by his student DeBakker [DeBak67]. It starts with the observation that many constructs in high level languages, Algol being taken as a canonical example, can be reduced to simpler ones either in that language or an allied language which does no violence to the original. For example, "a[3, 2]" can be reduced to "a[3][2]"; the conditional "a := if b then c else d" can be reduced to "if b then a := c else a := d"; the switch declaration "S := S1, S2" is replaced by "procedure S(n); if n = 1 then goto S1 else goto S2";

multiple labels are replaced by single labels, and so on. Such reductions are performed by a preprocessor. The use of such reductions to simplify the language which must be formally defined is a powerful and widely applicable technique, but not very profound.

The substantive portion of Van Wijngaarden's model is the processor which accepts the preprocessed text, scans and modifies it repeatedly, producing at each step the value of the text. "This value is a text which changes continuously during the process of reading and intermediary stages are just as important to know as the final value" [VanW63]. Stated more precisely, the input to the processor is a string consisting of the special operator value, followed by the preprocessed source text, followed by a set of rules which define the language in which the source text is written. The processor is an interpreter, table-driven by the language rules. These rules, which Van Wijngaarden terms "truths" are written in the metalanguage and fall into two classes:

(1) syntax rules, such as

⟨ identifier ⟩ in ⟨ simple variable ⟩

where "in" may be read as "is an element of the set"

(2) semantic evaluation rules, such as

value {⟨ sum 1⟩ + ⟨ term 1⟩} =

value {value ⟨ sum 1⟩ + value ⟨ term 1⟩}

which may be read roughly as: to obtain the value of the addition of a ⟨ sum 1⟩ and a ⟨ term 1⟩ begin by obtaining the value of the operands. It is interpreted as: if the string or a substring has the format specified by the left-hand side of the rule, replace it by the right-hand side of the rule, with appropriate handling of formal parameters.

The preprocessed source text, indeed any construction in the source language, is said to be a "name". The value of a name is obtained by consulting the list of evaluation rules until an applicable† evaluation rule is found and applying it, producing a new text. If the operator value appears in this text, the process repeats. The process of finding an applicable rule and applying it, of course, invokes a complex string scanning algorithm.

The crux of this method is that the execution of an evaluation rule can add new evaluation rules to the text. Since the source text acts as data and the language rules act as program of the processor, this scheme uses a program which grows as it runs. For example, assignment statements are handled by the semantic rule

value {⟨variable 1⟩ := ⟨expression 1⟩} =

{⟨variable 1⟩ = value ⟨expression 1⟩}

Hence, upon encountering a text containing

a := 3 + 4

the processor replaces this fragment by

a = value {3 + 4}

and, since the semantic rules include those for integer addition, ultimately by

a = 7

Note the change from the operator ":=" which is an operator of the source language to the operator "=" which signifies an evaluation rule in the meta-language. Subsequent references in the source text to "a", such as "a+b", will make use of this new truth in obtaining its value.

---

†The applicability of an evaluation rule is determined by the syntax rules which test for set membership.

There turns out to be one central difficulty with this proposal. The recording of all values as "truths" in a single string is perhaps elegant, but far too simple-minded to serve as a good representation. The lack of organization and the incredibly recursive fashion in which applicability of rules must be determined renders even the simplest example so unwieldy as to be incomprehensible. Also, because the representation is clumsy, the semantic metalanguage becomes quite complex, for the process of consulting "truths" requires an involved pattern match. Finally, the poor representation forces the language definitions to be needlessly complex and anything but transparent. Van Wijngaarden's method amounts to definition by Markov algorithm. It should, however, be clear that string processing is a poor representation of program evaluation. What is required is a representation which exploits the structure of source programs.

McCarthy, [McCar62] and [McCar66], in taking this position introduced the notion of an abstract syntax. This he defined as a set of predicates each true of objects in its characteristic syntactic class (e.g., isterm(t) is true only of ⟨term⟩s) and selectors which given an appropriate syntactic construct select out one of its parts (e.g., forlist(t) selects the ⟨forlist⟩ out of an ⟨iteration statement⟩. The predicates and selectors are defined recursively. For example, consider

$$\text{isterm}(t) = \text{isvar}(t) \vee \text{isconst}(t) \vee (\text{issum}(t) \wedge \text{isterm}(\text{addend}(t)) \wedge \text{isterm}(\text{augend}(t)))$$

which corresponds to the BNF

$$\langle \text{term} \rangle ::= \langle \text{var} \rangle \mid \langle \text{const} \rangle \mid \langle \text{term} \rangle + \langle \text{term} \rangle$$

The use of such an abstract syntax is effectively equivalent to using a BNF syntax and having a parse tree of the source text, with the additional benefit

that alternative right-hand sides are given names.

To discuss the semantics of a language, McCarthy uses a state vector ξ, defined at any given time to be the set of current assignments of values to variables of the program. Two primitive functions "a" and "c" access the value of a variable in ξ, and map a state onto a new state with changed value for one of its variables. The result of executing a program π with initial state vector ξ in language $\mathfrak{L}$ is defined to be a new state vector $\xi' = \hat{\mathfrak{L}}(\pi, \xi)$ where $\hat{\mathfrak{L}}$ is the semantic function of $\mathfrak{L}$. $\hat{\mathfrak{L}}$ acts as an interpreter of the program π, using selectors and predicates to decompose π, using its sequencing rules to sequence through π, and using state vectors $\xi, \xi_1, \xi_2, \ldots, \xi'$ to record the values of variables in π.

McCarthy applies this technique to the specification of a very restricted subset of Algol 60, called "Micro Algol". For this simple language, $\hat{\mathfrak{L}}$ can be specified very neatly as a simple recursive function. The elegance of this specification is due in part to the simplicity of Micro Algol. In particular, the language has no block structure so that (1) the set of variables comprising ξ_i is constant and (2) control can be represented by a single statement number.

However, the McCarthy formalism does contain several significant techniques. A state vector has an intuitive appeal and far better models the variables of a programming language than a sub-string of Van Wijngaarden's "truths". Dealing with abstract syntax neatly dodges such issues as written representation of the language and parsing, which are thorny but peripheral problems. Also, abstract syntax can be used to bypass much of the preprocessing actions required by other representations, so that specification is that much more direct. Finally, the use of a programming language to express the semantic interpreter invites the development of a problem-oriented metalanguage tailored to such expression.

2.1.4 Interpreter Models Continued: The Vienna School

The most extensive attempt at developing such a language was carried out by the Vienna Laboratories of IBM in the course of creating a formal definition of PL/I. This description [Alb68a], [Alb68b], [Fle68], [Luc68a], and [Walk68], was "a major development effort within IBM to prepare a completely formal description of PL/I including both formal syntax and semantics" [Nich68]. There is no doubt that the work is a major effort: the formal specification runs to nearly 1000 pages plus several volumes of informal, explanatory discussion.

The method and metalanguage† developed for this task are perhaps the most significant body of work to date in the field of formal language specification. Since a complete formal specification of PL/I has been written in it, there is empirical evidence that it can be used in a large-scale effort. Further, in the absence of any other fully developed model, it may become a de facto standard. In the past year, ULD has been used ([Ger70], [Lee69], [Rey69]) for the formal definition of at least three languages in projects unconnected with the IBM/ULD effort.

The ULD authors distinguish [Luc68b] three components characterizing a formal definition model: (1) the base, (2) the design, (3) the metalanguage. We shall adopt this tripartition in examining ULD.

The base is a modification of the technique used by Landin in describing a mechanical evaluator for the λ-calculus (c.f. §2.1.2). An abstract machine for some language £ is defined by its two components:

---

†We will use the term ULD to refer to both the method and metalanguage; context will distinguish between the two.

(1) a set of machine states ξ_i,

(2) a state transition function Λ.

For any source program $\mathscr{P}$ written in $\mathfrak{L}$, there is some initial state ξ_o which properly represents $\mathscr{P}$. Application of the state transition function Λ to a state ξ_i yields a set of possible successor states. That is, in general, Λ is nondeterministic so that a computation is defined to be a sequence of states $\xi_o, \xi_1, \xi_2, \ldots, \xi_i, \ldots$ such that $\xi_{i+1} \in \Lambda(\xi_i)$. (If different computations on the same program produce different results then the value of the program is undefined.)

The design of ULD centers around the use of structured objects – finite trees with named components – to represent the machine states. The components of a state include its storage, its environment, the text being interpreted with its statement counter, various directories for variables, and the control. The components of control are a set of instruction names which refer to instruction definitions. In a state ξ, any instruction in the control is a candidate for execution. Hence, $\Lambda(\xi)$ is defined to be

$$\{\psi_{in}(\xi) \mid \psi_{in} \text{ is an instruction in the control of } \xi\}$$

An instruction may return a value, modify and in particular add to the control, or change any part of the state. As the computation progresses, program constructs are moved from the text into control, ultimately producing a result which is reflected in a change to storage.

The underlying metalanguage is a melange of the propositional calculus, conditional expressions, arithmetic operations, functional composition, and two special operators which manipulate structured objects (a selector and a constructor). Of greater interest to this study is the notation in which instruction definitions are written, the semantic metalanguage. This consists of the underlying metalanguage augmented by a number of special

notations ("paraphrase extensions" in our terminology) intended to facilitate the writing of instructions. Of particular importance is a set of special forms for manipulation of the state components.

A number of criticisms can be leveled at ULD. The restriction of structured objects to trees and hence the prohibition against sharing components is needlessly constrictive. It rules out explicit and natural representation of sharing, which is a basic notation in programming, and forces the use of clumsy substitutes. Also, the semantic metalanguage is not well-chosen, particularly in regard to its syntax. While it uses most of the familiar concepts of programming (e.g., conditionals, sequencing, assignment of results to objects, procedures, and procedure calls), it presents these familiar concepts in strange guises and represents them in perversely nonstandard notation. Further, there are a number of restrictions in the language which make it awkward to specify changes to the state (c.f. [Lee69]).

However, as a number of researchers have independently chosen to use it for the description of quite different languages (PL/I, APL, BASIC, and an experimental language designed by Reynolds), there is good reason to believe it has considerable merit. It would seem that the weakness of the PL/I formal definition project was not ULD but the project goals. The immense complexity of the formal definition is attributable to the complexity of PL/I, not to defects in the metalanguage.

2.1.5 Assessment

Considering the various models which have been proposed for the semantic specification of programming languages, the most striking characteristic is the diversity of formalisms which have been used as bases. The elementary theory of computability demonstrates the equivalence of such

dissimilar formalisms as Turing machines, Markov algorithms, canonical systems, and the λ-calculus. Empirical observation discloses that almost all have been used as the basis for some exercise in semantic specification of programming languages. That is, these devices not only have the same computational power but they can all be used to represent the meaning of similar languages. On the one hand, this lends a certain credence to the you-can-do-anything-on-a-Turing-machine school; on the other, it displays the school's essential weakness: the high anguish factor.

The primary requirement imposed on a semantic specification is that it describes the meaning of language constructs precisely and clearly. It is easy, unfortunately too easy, to find a formalism in which the semantics of programming languages can somehow be represented and specified precisely. Precision is not the issue, nor is computational power. The real issue is effective representation; i.e., choosing a representation which minimizes the anguish factor. We require representations which preserve intuitive notions of structure and meaning so that formal specification is direct and clear. Failure to achieve such representation inevitably leads to a useless excursion in the Turing tar pits.

Using the criterion of effective representation, the models based on formalisms such as the λ-calculus and Markov algorithms fare poorly. While the bases are simple, rigorous, and tractable, they provide a poor representation of simple concepts in programming. In general, attempts to apply existing formalisms, developed for other purposes, to semantic specification is a misplaced effort; while they may be useful in analyses or exegesis of certain portions of a language, they break down when the burden of a complete description is placed upon them.

In short, there is a very wide latitude in the possible formalisms on which a semantic specification can be based. There is a certain temptation to attempt adaptation and utilization of a classical mathematical basis. However, this temptation must be resisted. If a direct and clear definition is to be obtained, a formal basis should be chosen whose primitives have an intuitive content in terms of the primitives of programming and, conversely, which abstract the primitives of programming.

The second issue which emerges from this study may be stated as: description vs. design. As noted in the introduction, an ad hoc inhomogenous language will inevitably imply a clumsy, inhomogenous specification. Regardless of the elegance and power of the metalanguage employed, the specification in that metalanguage must explicate every wart of the language. Well-chosen notation can occasionally obscure these blemishes, but hiding them completely is impossible. When the language becomes large, design errors grow from warts to humps and the formal semantic specification becomes a 1000-page monstrosity. For this reason, it is often a misplaced effort to attempt the a posteriori formal semantic specification of an existing, fixed language. Frequently, the only consequence of such work is to bring into sharp focus language defects.†

Such observations inspire the alternative approach: using the semantic specification a priori, in the design phase. The specification changes roles from an after-the-fact description to a notational tool in which to formulate, express, judge, and thereby improve the design. The benefits of a fluent

---

†For example, De Bakker in discussing his model [DeBak67] observes that "several aspects of the semantics of ALGOL 60, which are of no essential importance, have complicated and lengthened the definition . . . considerably."

notation to the design process should be obvious. Further, the elegance and efficiency with which a language feature can be formally expressed becomes a key criterion in judging its worth. In consequence, it is to be expected that languages so designed will not only be more homogenous, and hence easier to learn and use, but will exploit more completely their own mechanisms and potential linguistic power.

One final point should be noted. While the disadvantage of two-stage explication was observed in connection with compiler models, the difficulties are not confined to compilers. Quite the same objection may be raised against Landin's definition of Algol 60 semantics by means of a translator into IAE. It is less the nature of the target language that causes difficulties than that there is any target language at all. Semantic models which deliver a second program to be run in a separate phase do not give a direct description of meaning. For this reason, one-stage, interpreter-based models tend to be far more satisfactory.

2.2 EXTENSIBLE PROGRAMMING LANGUAGES

In the few years since its origin, the field of extensible programming languages has seen an astonishing growth. This is due in part to the obvious utility of a complete working extensible language, in part to the insight such research yields concerning the foundations of programming languages, and in part to a tantalizing air of universality which pervades the concept.† Also, a substantial part of this growth can be attributed to

†The well-trained ear will detect the siren song of UNCOL in the background.

a bandwagon effect: work which several years ago would have been categorized as being in the field of translator writing systems is now advertized as extensible language research, with suitable shifts in emphasis. Finally, the notion of extensible languages appears to be an idea whose time is becoming ripe. We know how to build far better languages than those currently available. For various reasons, utilization of this knowledge is being channeled primarily into extensible programming languages. Hence, there is considerable internal pressure to produce extensible languages.

Regardless of cause, the field is overgrown with vegetation. A list of languages claiming to be extensible and proposals for such languages includes [Abr66], [Bell68], [Ben68], [Chea66], [Earl69], [Gal67], [Gar67], [Har69], [Irons68], [Jorr69], [Kay68], [Leav66], [MacLar69], [McKe66], [Mills68], [New68], [Olyn69], [Stand68], [vanW69]. The very number of such proposals would alone preclude any but a trivial examination of them all in the scope of this work. However, their number is not the only obstacle to a complete review. Many of the above proposals are incomplete. For example, some present a scheme by means of examples without ever supplying the requisite detail; an assessment must guess at how the general case is treated and how the scheme is to be implemented. Others, while more detailed, attack only one small facet of extensibility, leaving in doubt whether such an approach can be integrated into a programming language.† Still others, while complete programming languages,

---

†To appreciate the importance of such integration, it should be recalled that the messy parts of implementing Algol 60 arose from the interaction of different facets [McCar69]. Two or more features, proposed by different designers, each had a straightforward implementation; their union did not.

are either unimplemented or incompletely documented, making it impossible to assess their practicability. In short, for most of these proposals a critical review could not be sufficiently deep to justify its existence.

Instead of treating all the above projects in a brief and shallow fashion, we shall concentrate on two of them chosen to illustrate significant aspects of the field. Each marks the state of the art in one or more types of extensibility. Further, each is a complete language, so it is possible to examine how the extension mechanisms are embedded in a programming language. Finally, each takes implementation seriously: either a working implementation exists or an implementation is in progress.

For those many proposals this treatment omits, we refer the reader to a survey paper by S. Gerhardt [Ger69] and to the proceedings of an extensible languages symposium [Chris69] held in May, 1969.

2.2.1 IMP

Perhaps the most obvious type of extension is syntactic. For various applications areas, an almost endless number of syntactic forms can be profitably added to a language to allow succinct expression of common forms used in these areas. The point of an extensible language is, of course, that not all syntactic forms need be in the language at any given time. Assuming that the language's syntax is specified by a set of context-free productions, the syntax can be extended by the addition of new productions. If the parse algorithm for the language is syntax-directed and accepts† the new productions, then the parser is correspondingly extended.

---

†It should be noted that some syntax-directed analyzers work correctly on only some subset of the context-free grammars. In such cases, an extension is implementable only if its productions adjoined to the existing set are acceptable to the analyzer. Our experience with several such

This sort of syntactic extension is provided in the programming language IMP, together with a sophisticated facility for specifying the meaning of such extensions.

IMP [Irons68], [Irons70] is an extensible programming language designed by E. Irons which has been in practical, real-world use since 1967. It is atypical among extensible languages in that its intended domain is system programming, primarily on a CDC 6600. Indeed, its chief applications to date have been the writing of the IDA-CRD† time-sharing system and several versions of the IMP compiler. In several respects, it is specifically designed for this purpose. Operations in the language include machine code, depend heavily upon word length, and deal with such matters as register allocation. Hence, IMP operates at a much lower level than most "high-level" programming languages and is by no means machine-independent. However, while these characteristics limit the exportability of the language, they are largely orthogonal to the extension mechanism. Our concern is with the latter.

The compiler system used for implementing IMP is a development of Irons' syntax-directed compiler for Algol 60 [Irons61]. It is on this compiler system that the syntax extension mechanism is based. To compile a program, the source text is first parsed, using a refinement of the algorithm

---

analyzers suggests that most grammar restrictions will prove unacceptably annoying to the programmer. The rewriting of productions into acceptable form is difficult, tedious, and tends to obscure the structure of the grammar. Hence, an analyzer which works on unrestricted or very nearly unrestricted context-free grammars is required. The analyzer used in IMP is nearly unrestricted and appears to work very well in this regard.

†Institute for Defense Analysis, Communications Research Division, located in Princeton, New Jersey.

described in [Irons63]. The result of the first step is the generation tree of the program.

Associated with each syntax production, whether basic or programmer-defined, is a set of semantic actions which define the meaning of that syntactic unit. Subsequent to parsing, the generation tree is repeatedly traversed (order being top-to-bottom, left-to-right) as many times as desired; the successive traverses are designated times 2, 4, 6, ... For each syntactic unit of the generation tree, actions associated with the corresponding syntax rule are performed at the time specified for that action.

To take a very simple example, consider the definition of an assignment form which switches the values of two variables. The syntax is given† by

$$\langle \text{expression} \rangle \ ::= \ \langle \text{name} \rangle_a \ \leftrightarrow \ \langle \text{name} \rangle_b$$

which states that an ⟨expression⟩ may have the format "⟨name⟩ ↔ ⟨name⟩". The role of the subscripts "a" and "b" will be clear momentarily. Suffixed to this syntax rule is the semantic specification

```
means at time 2
    'begin local t; t := a; a := b; b := t   end'
```

This states that at time 2 (first tree walk) an ⟨expression⟩ of the form ⟨name⟩ ↔ ⟨name⟩ is to be replaced by the parse tree for the text enclosed

---

†The notation used in IMP is based on the small character set used in its implementation and is somewhat awkward. For the purpose of exposition, we have modified Irons' notation to make it more suggestive of Algol 60. Irons would write

EXPR :: (A , NAME) ↔ (B , NAME)

in single quotation marks, where "a" and "b" are to be replaced by the first and second ⟨name⟩s found in the source program. In general, subscripting a syntactic type with an identifier in the right-hand side of a production establishes that identifier to be a formal parameter of the production. The quoted text may be any legal expression in the IMP language. The above text should be clear once it is explained that "local t" establishes that t is a full-word variable local to the begin-end block.

As a minor digression, it should be pointed out that IMP allows simple extensions to be simply specified by allowing certain information to be omitted. In such cases, IMP automatically supplies a reasonable default value. For example, the time specification may be absent and will then be taken to be time 2. Similarly, in the syntax portion of a rule, the left-hand side may be absent and is then given default value ⟨expression⟩. Hence, the same effect as given by the above example can be obtained by writing

⟨name⟩$_a$ ↔ ⟨name⟩$_b$ means

'begin local t; t := a; a:= b; b := t end'

In general, wherever the system can make an intelligent guess as to the value of a field, that field is optional. This makes it easy for programmers of varying degrees of sophistication to use the extension facilities; the more a programmer knows about the facilities, the more power is available to him.

The above example is a case of macro processing in the classic sense, upgraded and applied to high-level languages with syntactic types. Such macro extensions are just like procedure calls in Algol 60 except that the syntax of procedure calls is fixed while each production may, in general, have a different syntactic shape. Note that the macro extension is

characterized by its simple substitution of actuals for formals in the semantic routine, not by in-line expansion of code. The choice between in-line code and common subroutine is an implementation issue.

To give this discussion proper perspective, it must be immediately emphasized that macro extensions are but the simplest type of semantic specification provided in IMP. Irons argues [Irons70] that while necessary, it is by no means sufficient; the macro type of extension is only the beginning of a complete system.

> Without embellishment, . . . this description method admits only a very limited treatment of semantics. . . . In various efforts to overcome the deficiencies of the simple macro process for semantics, additional embellishments of varying complexity have been proposed for the semantic portions of extensions such as those of IMP. The author's experience with some of these experiments in specification finally led him to the conclusion that nothing short of a general programming language capable of operating on syntactic and semantic structures would be adequate to express even the moderately difficult concepts required in the translation process. The realization of this in IMP is to consider that the semantic part of an extension is in fact not a macro shell but a computation which is evaluated as part of the translation process. The computation could be expressed in any suitable language, but for economy of notation, IMP was chosen for this purpose.

An example may make clear the notion of computing the semantics of an extension. Consider defining a simple iteration form which proceeds from a lower limit to an upper limit in steps of 1. Suppose we wish this loop expanded into straight line code whenever the number of iterations is a manifest constant and smaller than some compile-time variable k.

⟨expression⟩ ::= for ⟨name⟩$_i$ ← ⟨expression⟩$_\ell$ to ⟨expression⟩$_u$ do
⟨expression⟩$_{body}$

means at time 2

```
if constant(ℓ) ∧ constant(u) ∧ convert(u) - convert(ℓ) < k
then expand (body , i , ℓ , u)
else 'begin local tag;  i ← ℓ
                tag : body;
                if  i ≠ u then begin  i ← i+1 ; go to tag end
     end'
```

The semantic specification tests at compile-time to see if the form is a candidate for expansion. If so, it calls a function expand which returns the form

'begin i ← ℓ ; body ; i ← ℓ +1 ; body ; . . . i ← u ; body end'

where the number of instances of body is u - ℓ + 1. If, however, the form cannot be expanded (say, because the upper limit is not a constant) then the semantic specification produces a block containing the appropriate loop.

In general, the semantic actions associated with a production can invoke arbitrary compile-time computation. Since these computations have access to the entire compiler mechanism, any extension whatever is possible. The real issues are the ease with which extensions can be made, the clarity of their specification, and the degree of implementation independence. What is required is an interesting class of compile-time actions which are more sophisticated than macro semantics but which do not require a detailed knowledge of the compiler for their specification.

IMP goes some distance in providing such a class of actions. Pattern-matching operations which act on trees are provided, so that it is possible to

easily discriminate sub-cases of a syntactic construct, e.g., for separate handling of special cases. Also, it is possible to associate data types with syntactic types in productions. For example, the production

$$\langle \text{expression} \rangle ::= \langle \text{expression} \rangle_{\text{a, real}} \div \langle \text{expression} \rangle_{\text{b, int}} \text{ means } \ldots$$

will be applicable to a fragment of source program only if the first and second operands are of types real and int, respectively. This has two useful consequences. Type information is brought into the semantics in an implementation-independent fashion; hence, a change in such matters as table structure will not affect the validity of extension definitions. Also, the syntax and data type specification are largely orthogonal, keeping each small and simple.

2.2.2 Algol 68

The algorithmic language Algol 68 [vanW69], [Lind69] provides an instructive contrast to IMP. Whereas IMP has well-developed facilities for syntax extension, Algol 68 has only the weakest. Whereas IMP generally operates at a lower level than conventional algorithmic languages, dealing with machine words and admitting machine code, Algol 68 is formulated in terms of a "hypothetical computer" which physical implementations may "model". Further, IMP is fairly simple and must be extended to obtain sophisticated forms, whereas Algol 68 comes to the programmer as a fully developed (and in some respects unmodifiable) language. Finally, IMP is generally weak in its provision for data types, whereas data types, their definition, and their interaction are among the chief concerns of Algol 68.

Algol 68 employs three principal extension mechanisms: builtin, data type, and operator. We consider these in turn.

The builtin extensions are used to simplify the semantic specification of Algol 68. Unlike IMP, which is defined mechanically and rigorously by its syntactic/semantic productions, Algol 68 is defined by a written document, Report on the Algorithmic Language Algol 68 [vanW69]. To minimize the number of and keep orthogonal the primitive concepts, an austere and somewhat unnatural language kernel, termed the strict language, is defined as being basic. Its semantics are specified by a quasi-English description. The Report then goes on to define an extended language in terms of the strict language.

The strict language contains all semantic concepts; the extended language allows convenient forms of paraphrase for many constructs and notions. These paraphrases are designed principally to (1) abbreviate commonly occuring forms, (2) make (extended) Algol 68 resemble the familiar Algol 60 as much as possible, (3) enhance readability of code. For example, the strict language has no iteration statement; the extended language defines several iteration forms in terms of appropriate loops written in the strict language. Similarly, a case statement is absent in the strict language and defined in the extended language in terms of a strict language conditional. Mimicking of Algol 60 occurs frequently; for example, the extended language declaration

real x;

may be used instead of the equivalent strict form

ref real x = loc real;

These builtin extensions differ from the extensions treated elsewhere in this paper in that the programmer has no hand in them. They are strictly for the authors of the defining Report. Hence, as Algol 68 is currently constructed, its builtin extensions properly fall outside the scope

of "extensions", in the sense with which the term is generally used. However, there is a close relation. To a large extent, the builtin extensions take the form of macro extensions as discussed in section 2.2.1. With suitable additions to the language, it would be possible to allow programmer-defined extensions having the same form as the builtin extensions now provided, thereby obtaining a rudimentary facility for syntax extension.

In Algol 68, the notion of data type is denoted by the term "mode". Five modes are primitive: bool (i.e., Boolean), int (integer), real (floating point), char (character), and format (input/output format). From these primitive modes, other modes can be defined, using five classes of formation rules:

(1) definition of pointer (i.e., ref) types. For example, "ref real" is the mode of objects which can point to reals.

(2) structures (i.e., structs) much like those of COBOL [COBOL61] or PL/I [IBM66]. For example,

    struct (real price, char code, ref int invoiceaddress)

    is the mode of structures having three components: a real designated as "price", a char designated "code", and a ref int designated "invoiceaddress".

(3) arrays of objects all the same mode. For example,

    (a) "[1:m, 1:n] real" is the mode of two-dimensional m by n arrays of reals.

    (b) "[1:0 flex] char" is the mode of one-dimensional arrays of chars with lower bound 1 and upper bound flexible. (That is, for such an object, say x, x[i] may be assigned for any

positive i. The system will insure, e.g. by dynamic storage allocation, that there is a storage location available.)

(4) the union of other modes. For example,

union (int, char, struct(real re, real im))

is the mode of objects which can vary between being ints, chars, and struct(real re, real im)'s.

(5) the definition of procedure types. For example,

proc(real, union(int, char)) ref bool

is the mode of procedures taking two arguments, a real and either an int or a char, and delivering a pointer to a bool.

To be really useful, the creation of new modes implies the creation of operations which act on values of these modes. Procedures can be defined as in Algol 60, but Algol 68 goes one step further and allows definition of new operators – uniary prefix and binary infix. An operator is defined by specifying its symbol (e.g., +, ↑, abs), its formal parameters with their types, the mode of the result, and a defining body like that of a procedure. In addition, a binary operator has a priority number – an integer between zero and ten – which specifies its binding strength relative to other operators. Operators differ from procedures in that a given operator may simultaneously have many different definitions for different modes of formal parameters. The compiler selects the appropriate definition based on the modes of the actual operands.

The notion of mode in Algol 68 is subject to one important restriction: all transactions with modes are carried out at compile-time. Hence, modes are not values which can be computed, but rather attributes which are processed statically. This is, of course, a natural outgrowth of

Algol 60 and a plausible restriction if a compiler-based system is assumed. However, it rules out certain generality in the language and imposes the restriction that all variability in data type be explicitly spelled out when the program is written. We return to this issue in section 7.1.6.

Section 3. INFORMAL DESCRIPTION OF EL1

It was concluded in section 2.1.5 that, for our purposes, the semantics of a programming language is best specified by an interpreter model. The interpreter takes as input a suitable representation of a program and produces as output a suitable representation of the value of that program. In carrying through this approach, two major issues must be settled: (1) how programs will be represented in the interpreter, (2) in what language the interpreter will be written.

It is our thesis that the defining interpreter should be written in the language being defined.† This implies that programs are to be represented as data objects in that language. If this language is extensible and has an adequate data type definition facility, the representation presents no problem: it is necessary only to define a set of data types which represent programs and their components.

We have designed a language, EL1, intended to serve as the base for an extensible language and have specified its syntax in this manner. In this section, the language is discussed informally. English description and numerous examples are used to give the reader sufficient fluency in EL1 that he can read code. In section 5, a formal definition of EL1 is given: its syntax in BNF and its semantics by an EL1 interpreter. Hence, this section serves both as an introduction to the language and as an explanation of the notation used in its formal definition.

---

†It will be argued by some that the definition of a language by a program in that language constitutes a circularity which renders the definition useless or logically invalid. We contend that such arguments are wrong and address the issue in section 4.2.

Because it serves as an expository introduction, this section does not attempt either formal precision or completeness. Terms are occasionally used with only an informal definition, or with merely an appeal to the reader's intuition. In all cases, a precise definition is given in the formal specification of section 5. As an aid to cross-referencing between the informal and formal specification, subsection 3.i corresponds directly to 5.i for i = 2, 3, ..., 12. To keep this section from growing to unwieldly size, the discussion is aimed almost exclusively at presenting the language, not justifying it. The rationale behind language features, analysis of these features, and comparison with other languages are carried out in section 7.

We should, at this point, stress that ELl as it currently stands is only the base of an extensible language. It is not of itself a complete language core. For example, it does not contain facilities for syntax extension. Such facilities are well understood; their treatment here would diffuse and thus weaken this work. In section 9, we outline the additions which must be made to the present design to obtain a complete core.

3.1 INTRODUCTION TO EL1

Since EL1 is designed as an extensible base it contains, at least in germinal form, most of the notions found in standard programming languages. In particular, it draws upon Algol 60 [Naur63], Lisp 1.5 [McCar62], and Algol 68 [VanW69]. Most of the concepts found in these three languages are present in EL1 – sometimes in changed notation, occasionally generalized, often simplified. EL1 is perhaps best introduced by comparison with these three existing languages.

As in Algol 60, variables in EL1 are declared to be of some data type.† A variable may be changed by assignment but is restricted to values of the declared type. Operators and procedures take arguments of appropriate types and produce values of appropriate types. The notation for assignment, infix operators, and procedure calls is almost identical to that of Algol 60. A conditional is provided, as is a form for iteration similar to the Algol ⟨for statement⟩.

From Lisp 1.5, EL1 borrows two principal notions. (1) The syntax is particularly homogenous. It is arranged so that almost every construct is of syntactic type form: including assignments, procedure calls, conditionals, and expressions formed from infix operations. Every form has a value. With only a few exceptions, any construct used anywhere in the

---

† The declaration of a variable in any programming languages may serve two distinct purposes:
(1) specifying that the variable is local (as opposed to being a free variable),
(2) specifying the data type of the variable.
In a language which admits both local and free variables, (1) is unavoidable. However, if the language has the notion of a default data type, (2) may be suppressed; it would be a simple extension to add this facility to EL1.

language may be replaced by a form having the same value. (2) Storage allocation is not confined to a stack. In Algol 60, the only classes of objects admitted in the language were those whose storage allocation could be held completely on a stack. Lisp, on the other hand, deals with lists whose size may vary during execution. Storage for these lists comes from a region, list-structure space, out of which allocations are made at run-time – one block for each cons. In contrast to Algol 60 variables, lists are not necessarily destroyed on the exit of the procedure in which they are created; hence, storage blocks must be reclaimed by garbage collection. EL1 provides for two classes of objects: (1) those which behave like Algol 60 objects and are implemented on a stack, (2) those which behave like Lisp objects and are implemented by dynamic storage allocation with garbage collection.

In this respect, EL1 is like Algol 68 which also has both stack and dynamic storage allocation. It also resembles Algol 68 in another important feature: the set of data types in the language is not fixed. New data types, or modes, can be defined in terms of existing modes using several builtin formation rules. Once defined, a new mode may be used exactly as if it were primitive, for example in declaring the data type of variables or in defining still other modes.

One significant difference of EL1 and Algol 68 is that in EL1 modes are treated dynamically, i.e., the definition of a new mode is an executable form, while in Algol 68 modes are treated statically. (The EL1 method is explained in section 3.9 and compared with that of Algol 68 in section 7.1.6). EL1 differs from Algol 60, Lisp 1.5, and Algol 68 in a variety of other ways, principally in certain omissions made for the sake of simplicity. For example, EL1 has the data type integer but not real. It is not our contention

that the latter should be defined in terms of the former; the widespread availability of floating point hardware dictates that reals be primitive. Rather, we argue that explanation and description of the language is somewhat simplified by its omission. Reals can later be added to the language with no violence to its structure.

3.2 CHARACTER SETS AND THE REFERENCE LANGUAGE

Since this description makes extensive use of examples, a note on character sets and written representation is in order. Algol 60 distinguishes three different "levels of language": a Reference Language, a Publication Language, and several Hardware Representations. These define three classes of written representation for Algol 60 programs, differing in character sets and related by simple transformations.† EL1 has two "levels" of language: a reference language and several hardware representations.

This paper uses the former. It has been chosen for ease of writing, reading, and typing. Upper case letters replace the boldface letters of Algol 60; for example: FOR, WHILE, END, and TRUE instead of **for**, **while**, **end**, and **true**. Algol 60 has three fonts of the Roman alphabet – lower case boldface, lower case normal, and upper case normal – and reserves the boldface font for forming ⟨basic symbols⟩. EL1 is restricted to two fonts – upper case and lower case – and uses upper case for its special symbols. For example, BEGIN is a delimiter while begin is

---

†With few exceptions, the transformations may be carried out by simple finite state transducers.

a variable. Where frequency of use warranted such addition, the Algol 60 character set has been supplemented to allow improved notation. For example, assignment is indicated by a left-pointing arrow (x ← y) instead of colon-equal (x := y). However, where there were no strong contrary considerations, the notation used in Algol 60 has been carried over. For example, "f(x, y + 1)" is the application of procedure f to the arguments x and y+1; "d[i]" is the i^{th} component of the object d.

Little will be said concerning hardware representations. They are provided as a concession to the possibility that a given implementation will not have available the full character set used in the reference language. Most likely to be missing is a second alphabetic font or some of the special characters. Any well-defined method which encodes the desired character set into the smaller one is acceptable. For example, special symbols such as BEGIN may be designated as reserved words. Alternately, special symbols may be written in some distinctive fashion, e.g., BEGIN!, .BEGIN., or 'BEGIN'. Clearly, such encodements will be somewhat annoying to use and will make code more difficult to read. However, these difficulties are the inevitable consequence of a restricted type font. (The skeptical reader is invited to rewrite a paper in automata theory using a 48-character set.) We believe that such considerations argue for expanded character sets on I/O devices. The notion of hardware representation is but a makeshift arrangement to serve in the interim.

3.3 PROGRAMS AND FORMS

The basic unit in EL1 is the form. Forms include:

(1) constants such as 13 and TRUE,

(2) variables such as x and pressure,

(3) infix operations such as x+y and i-j*k,

(4) selection of part of a compound object such as b[i] and position[3*x],

(5) procedure calls such as f(x) and foo(i, j+k, a[n]).

A form is a syntacticly complete unit and has a value. Forms may be put together in accord with the various composition rules of the language to obtain larger forms.

A program is simply a form which is not part of a larger form. Other than this, it is in no way special; it is evaluated according to the same rules as any other form.

3.4 CONSTANTS AND BUILTIN DATA TYPES

EL1 has ten builtin (i.e., predefined) data types: Boolean, integer, character, mode, ptr-any, procedure, none, noneref, symbol, and stack. For each data type† there are constant values of that type and builtin representations for these values. The data type of all constants is manifest, i.e., the data type of a constant can be determined uniquely from its written representation.

Booleans and integers, called bools and ints in EL1, are similar to their counterparts in Algol 60. The latter are identical in meaning and written representation to Algol 60 ⟨integer⟩s, e.g., 1, -5, 1596, 6600. Booleans in EL1 differ from those in Algol 60 only in their written representation: TRUE and FALSE instead of **true** and **false**.

---

†A single exception being the data type STACK; constants of this type would be of little use – c.f. §3.16.3.

The set of characters defined in the reference language includes the ten digits, an upper and lower case Latin alphabet, and a number of special characters. A character constant is represented by prefixing a single quote to a character. For example, the character B is written 'B, the character = is written '=, and the character ' is written ''.

EL1 allows the definition of new data types or modes. Hence, it is necessary to deal with modes as values. It will be recalled that the mode of 31 is int and the mode of TRUE is bool. Similarly, the mode of a data type is mode. That is, suppose the value of some form $\mathcal{F}$ defines a new data type; this value must have a mode. It does: the mode of this value is mode. Of the ten builtin modes, seven are primitive. Each of these is denoted by a mode constant: INT, BOOL, CHAR, NONE, NONEREF, PTR-ANY, and STACK.†

Having described the modes bool, int, char, and mode we have enough data types to begin discussion of the language. The remaining six modes will be discussed later when sufficient background has been developed to motivate their presentation.

3.5 IDENTIFIERS AND SIMPLE DECLARATIONS

An identifier is the name of a variable. It will be recalled that a variable has a fixed mode which is determined by declaration. Identifiers and the variables they name can come into existence in two ways: as formal parameters or declared variables. For brevity we here discuss

---

†If this is unclear, an analogy may be helpful. TRUE is a Boolean constant; its mode is bool. CHAR is a mode constant; its mode is mode.

only the latter case. Consider the EL1 fragment

```
DECL i, j, temp : INT;
```

This creates three variables named "i", "j", and "temp" each having mode INT. (Recall that INT is a constant which denotes a primitive data type.) Since the mode of i is INT, it can hold an INT value†; hence, the following assignment is legal

```
i ← 1079;
```

Further, since i holds an INT value, this value may be assigned to any other object whose mode is INT, e.g.,

```
temp ← i;
```

Analogously, variables of mode BOOL and CHAR may be declared and given values

```
DECL b1, test : BOOL;
DECL c, first : CHAR;
b1 ← TRUE;
test ← b1;
c ← 'q;
first ← c;
c ← '+;
```

The variables b1 and test now have Boolean value TRUE, the variable first has character value 'q, and the variable c has character value '+.

---

†Note that this differs from the rules of strict Algol 68.

An identifier is written[†] as a sequence of characters, the first being lower case Latin and the rest being either lower case Latin, digits, or the special characters "ρ" and "_". For example:

i, j6, temperature, intρ, p26street.

An identifier ends with the last character preceding a blank or special character such as ← or +. Since the minus sign is a special character, hyphenated names are written using the connector character "_"; for example:

overflow_flag, master_file, bind_formals.

3.6 BINARY OPERATIONS

Various binary operations are defined in EL1, many having meanings similar to Algol counterparts. Four binary operators take arguments of mode INT and deliver results of mode INT; these are: +, -, *, and /. Six other binary operators take INT arguments and yield BOOL results: ≥, >, <, ≤, =, and ≠. For example, consider

DECL i, j, k, l : INT;

The following are legal EL1 forms with modes INT, INT, BOOL, and BOOL, respectively

i + j, i + j * k * l, i > j, i ≤ k + j

EL1 differs from Algol 60 in one respect: all infix operators have the same precedence (and all associate to the right).[‡] Any meaning other than this must be explicitly indicated by use of parentheses. For example,

---

[†]Properly speaking, the above format is only one of two possible formats for identifiers, c.f. §5.5.1. The introduction of the second format at this point would, however, only obscure the discussion.

[‡]This is the same convention as that used in APL [Iver62].

```
j + k * l     means     j + (k * l)
```

but

```
j * k + i     means     j * (k + i).
```

To get the Algol 60 meaning in the latter case, one would write (j * k) + i. Giving all operators the same precedence is not advocated as a necessary principal of language design; it may be useful in some applications but quite awkward in others.† It is used in EL1 only to keep the syntax as simple as possible. We assume that in some cases one of the first extensions to EL1 would be a hierarchy of precedence levels in the style of section 3.3.1 of the revised Algol report [Naur63].

Two operators take BOOL arguments and deliver a BOOL result: "∨" (meaning logical OR) and "∧" (meaning logical AND). These evaluate their arguments only so far as necessary to obtain a result, e.g., if the first argument of "∨" is TRUE, the second is not evaluated. Hence, side effects may be different from those of similar Algol 60 forms.

The operators "=" and "≠" are not restricted to INT arguments. They accept operands of any builtin mode and return a BOOL result: TRUE iff the operands are of the same mode and identical objects of that mode. For example, consider

```
DECL c1, c2 : CHAR;
DECL b1, b2 : BOOL;
c1 ← 's;  c2 ← c1;
b1 ← FALSE;  b2 ← FALSE.
```

Then "b1 = b2" has value TRUE, "c1 = b2" is FALSE, "c1 = c2" is TRUE, and "b1 = FALSE" is TRUE.

---

† For example, a hierarchy of precedence levels has proved quite useful in Fortran and Algol 60 where the number of operators is small and fixed. On the other hand, consider APL which has additional data types and has a large number of operators. It would be almost impossible for a programmer to remember any precedence hierarchy of all the APL operators. Hence, APL uses the single,easily remembered rule that all operators have the same precedence.

Two other binary operators are less conventional. As mentioned earlier, the application of a procedure to its arguments can be denoted in the style of Algol 60, e.g., "f(x)". When the procedure takes a single argument, this can also be denoted as a binary operation, the operator being a small circle, e.g., "f∘x". This notation proves most useful in denoting a sequence of unary function applications. Instead of "f(g(h(p(x))))" one can write "f∘g∘h∘p∘x". Mixing the two notations, one obtains the standard usage "f∘g∘h∘p(x)".

Assignment is also treated as a binary operation: taking as right operand an object whose value becomes the new value of the left operand. The rule that all binary operators associate to the right makes forms such as

```
s ← x + y    or    i ← j + k * l
```

have their conventional meanings. Since an assignment is a binary operation, it has a value: its right-hand operand. Hence,

```
k ← 5 * (j ← k + 15)
```

adds 15 to the value of k, stores this in j, multiplies this by 5, and stores the result in k.

3.7 COMPOUND FORMS

In carrying out an algorithm, it is generally useful to evaluate a set of forms, with control passing from one form to the next according to some sequencing rule. Familiar sequencing rules found in programming languages include the following.

(1) statement sequencing in Algol 60:

After evaluating statement n, statement n+1 is evaluated (unless statement n contained an executed go to).

(2) the conditional expression of Algol 60:

If the ⟨Boolean expression⟩ is true then one ⟨arithmetic expression⟩ is evaluated, otherwise the other ⟨arithmetic expression⟩.

(3) the conditional expression of LISP 1.5:

Successive elements of a sequence of predicates $p_1, \ldots, p_n$ are evaluated in turn until one, say p_i, is found with value TRUE. The value of the conditional is then the value of the associated expression, e_i. If no p_i has value TRUE, the value of the conditional is undefined.

In EL1, these syntactic constructs and their evaluation rules are unified into a single form: the compound form. A compound form consists of a sequence of statements separated by semicolons and surrounded by the delimiters BEGIN and END. A statement is either a form or a clause, where a clause has the format

form ⇒ form

(The double-shafted arrow which separates the two forms of a clause may be read roughly as a causation or implication sign.) For example, the following compound form contains three statements: the first and third being forms and the second being a clause

```
BEGIN
i ← i + 1;
i = k ⇒ i + 2;
j ← 5 * i;
END
```

A compound form is evaluated as follows. Its statements are considered in turn, starting with the first. There are two cases. (1) If the statement is a form it is evaluated. Following this, the next statement is considered; if there is no next statement, evaluation of the compound form stops and the value of the compound form is the value of the last form evaluated. (2) If the statement is a clause, its first form is evaluated, producing a result $\mathcal{T}$ which is expected to be a BOOL. If $\mathcal{T}$ = TRUE, then the second form is evaluated, producing a result $\mathcal{V}$; evaluation of the compound form stops and $\mathcal{V}$ is its value. If, however, $\mathcal{T}$ = FALSE, then the next statement is considered.

For example, the above compound form is evaluated in the following steps.

(1) i is increased by 1.

(2) If i = k then the value of the compound form is i + 2 and evaluation stops.

(3) Otherwise, 5 * i is assigned to j and this is the value of the compound form.

To facilitate writing, two syntactic variations in a compound form are allowed.

(1) The bracketing pair BEGIN, END may be replaced by the pair [[,]].

(2) The semicolon preceding the terminal bracket may be omitted. For example, the above compound form may be written

$$[\![\, i \leftarrow i + 1;\quad i = k \Rightarrow i + 2;\quad j \leftarrow 5 * i \,]\!]$$

Three special cases of the compound form are of interest.

(1) If all the statements are forms, then we have the ⟨compound statement⟩ of Algol 60:

BEGIN s_1; s_2; . . . s_n END

where the s_i are forms.

(2) If all the statements are clauses, we have the conditional expression of Lisp 1.5:

$$[\![p_1 \Rightarrow e_1 ; p_2 \Rightarrow e_2 ; \ldots p_n \Rightarrow e_n]\!]$$

where the p_i's are predicates and the e_i's are arbitrary forms.

(3) If there are precisely two statements, the first of which is a clause and the second of which is a form, we have the if-then-else of Algol:

$$[\![p_1 \Rightarrow e_1 ; e_2]\!]$$

3.8 ITERATION

As in any algorithmic language, it is frequently necessary to evaluate a form repeatedly, possibly with an index variable changing on each repetition. The EL1 iteration form is provided for this purpose. It differs from its Algol 60 counterparts in a few respects, most notably in its syntax. For example, the following fragment computes the sum of the squares of the positive odd integers less than or equal to n

```
s ← 0;
FOR i ← 1, 3, ..., n DO s ← s + i * i;
```

In an iteration, the first form following the arrow is the initial value of the index variable; the step size is the difference between the second and first forms; the third form is the limit. The index variable is a new variable, of mode INT, which is local to the iteration. That is, the i above has no relation to any other variable named i which might exist in the program of which the above fragment is a part.

If the step size is to be 1, the second form may be omitted. For example, the following fragment computes the n^{th} Fibonacci number†

† The Fibonacci numbers form a sequence 0, 1, 1, 2, 3, 5, 8, 13, ... in which each number is the sum of the previous two. They are defined by the relations

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2} \quad (n \geq 2)$$

for $n \geq 2$, leaving the result in the variable "fib":

```
old ← 0;
fib ← 1;
FOR i ← 2, ..., n DO [[ temp ← fib; fib ← fib+old; old ← temp ]]
```

If the index variable initially exceeds the limit, then the iteration is not executed. Hence, the above fragment actually computes the n^{th} Fibonacci number for $n \geq 1$.

Occasionally, it is desirable to terminate an iteration loop before the index variable has reached the limit. That is, an iteration proceeds until its limit is reached or TILL[†] some test condition becomes true. For example, the form

```
FOR i ← 1, ..., n TILL f(x) DO x ← g(i, x)
```

repeatedly assigns to x the value of g applied to i and x for i in the sequence 1, 2, 3, ..., n except that if f(x) is ever TRUE, the iteration halts.

A common variation is to carry on an iteration while some form has the value TRUE. One could write

```
FOR i ← 1, ..., n TILL not ∘ f(x) DO x ← g(i, x)
```

but the intention is far clearer if the double negative is eliminated. Hence, the equivalent form

```
FOR 0 ← 1, ..., n WHILE f(x) DO x ← g(i, x)
```

is admissible; g of i and x will be repeatedly assigned to x for each i in the range 1 to n, so long as f(x) remains TRUE.

[†] EL1 uses the delimiter TILL to avoid confusion with the Algol 60 **until**; the two have somewhat different uses.

With both the TILL and WHILE modifiers, the test condition is evaluated immediately before each evaluation of the form being iterated. If the value of the test calls for stopping the iteration, evaluation of the iterated form is inhibited.

3.9 MODE-VALUED FORMS

Those aspects of EL1 discussed thus far do not differ significantly from Algol 60 or, indeed, any other algorithmic language. Notation and syntax have been somewhat idiosyncratic but the underlying semantics and facilities provided have been quite conventional. However, having explained our notation for familiar concepts, we now have sufficient foundation to present the innovative aspects of EL1. Most important of these is mode-valued forms.

3.9.1 Mode-Valued Constants and Identifiers

As discussed in section 3.4, the term mode is used in EL1 to designate a certain class of objects: objects corresponding to the intuitive notion of data type. Seven modes are primitive and denoted by mode-valued constants: INT, BOOL, CHAR, NONE, NONEREF, PTR-ANY, and STACK.

Just as variables can be declared of type INT and are thereby restricted to INT values, variables can be declared of type mode and are restricted to mode values. For example,

```
DECL m1, m2, complex, rational : mode;
```

The variables m1, m2, complex, and rational are mode-valued variables. Hence, it is legal to assign

```
m1 ← BOOL;
m2 ← m1;
```

The variables m1 and m2 now have the same value as the constant BOOL. Hence, "m2 = BOOL" is TRUE while "m1 = CHAR" is FALSE.

Of themselves, mode constants and mode-valued variables are uninteresting. However, we next consider operators which take modes as operands and produce new modes. Four such mode-producing operators are predefined: ROW, STRUCT, PTR, and RANY. From these, other mode-valued forms can be synthesized using conditional expressions, functional composition, and recursion.

3.9.2 ROW

The operator ROW is best introduced by means of an example. Consider the fragment

```
DECL triple : mode ;
```

The variable triple has been declared to be an object whose value is a mode.

```
triple ← ROW(3, INT);
```

The variable triple now has a value; i.e., the mode integer-arrays-of-length-three. Hence, it is possible to later write

```
DECL x, y : triple ;
```

The variables x and y are of mode triple. This corresponds roughly to the Algol 60 declaration

```
integer array x, y [1 : 3];
```

As a consequence of the EL1 declaration, x and y are objects having several of the properties of an Algol 60 array. They can be subscripted, e.g., x[2], y[i], y[f(x[i])]. The result of the subscripting operation is an object of mode INT which may possess a value and which may be changed by assignment. For example,

```
x[1] ← 10;  x[2] ← 20;  x[3] ← 30;
y[1] ← x[3] - x[2];
```

An EL1 variable of mode triple differs from an Algol 60 variable of type integer array in one primary respect. The Algol y can appear only as a ⟨subscripted variable⟩ or ⟨actual parameter⟩, these being the only two uses of arrays. The EL1 y can serve as operand for various operators; in particular, for the assignment operator. For example,

```
y ← x;
```

is legal and assigns to y[1], y[2], and y[3] the values of x[1], x[2], and x[3], respectively. This is a copying of values, not sharing. For example, if we next assign

```
x[2] ← 79;
```

this does not change y[2] which remains 20.

We say that ROW(3, INT) is an EL1 form having a mode value and that this mode is of class row. An object (such as y) whose mode is of class row† has a number of properties.

(1) It is composed of a sequence of identical components (e.g., x consists of 3 components each having mode INT).

(2) Any one of these components may be selected by subscripting (e.g., x[1] is the first of the 3 INTs).

(3) The number of components may be determined by applying the function length to the object (e.g., length(x) is 3).

(4) Given two such objects x and y, the form "x ← y" assigns the value of y to be the new value of x.

---

†It will be useful to abbreviate this notation and say, for example, "y is a row", meaning that y is an object whose mode is of class row.

It is frequently useful to define a mode of class row in which the number of components is not bound at the time the mode is created. For example,

```
DECL string: mode;
string ← ROW(CHAR);
```

This first declares string to be a variable of data type mode and then assigns to string the mode intuitively described by: array of any number of characters. Since the number of components in a string is not bound, string is said to be length unresolved. This is not to say that objects of mode string have variable length, but rather that the mode string leaves the length open. A specific object of mode string will have some fixed length, but different strings may have different lengths.

When a variable is declared to be a string, it is necessary to resolve the mode by specifying a length, i.e., a SIZE. For example,

```
DECL s : string SIZE ⟨n⟩;
```

declares s to be a string variable whose length is the current value of n. (The significance of the angle brackets around the length specification will become clear in section 3.11.) Subsequent to declaration, s behaves as any other object whose mode is of class row: it has a fixed number of components which may be obtained by "length(s)"; it may be subscripted; it may be changed by assignment.

To summarize, the operator ROW takes either

(a) one argument, a mode $\mathfrak{M}$, and then produces the mode: length unresolved sequence of components each of mode $\mathfrak{M}$,

(b) two arguments, an integer $\mathscr{S}$ and a mode $\mathfrak{M}$, and then produces the mode: sequence of $\mathscr{S}$ components each of mode $\mathfrak{M}$.

In general, the arguments of ROW may be arbitrary forms, provided that they evaluate to objects of the appropriate type. For example,

```
funny_mode ← ROW(⟦i > j ⇒ i; 10 * j⟧,
                 ⟦p(x) ⇒ INT; q(x) ⇒ triple; CHAR⟧)
```

It should be noted that ROW, like the other three mode-valued operators, is treated as a procedure: its arguments are evaluated, its body is executed, and it delivers a result whose data type is mode. One consequence of this is that the output of ROW can be used as the argument to a second evaluation of ROW, e.g.,

```
bool_matrix ← ROW(ROW(BOOL));
```

The mode bool-matrix corresponds to the notion of a two-dimensional array of Booleans. Since length has been specified for neither evaluation of ROW, bool-matrix is length unresolved with two unresolved dimensions. Any declaration of a variable to be a bool-matrix must specify both dimensions, e.g.,

```
DECL b : bool_matrix SIZE ⟨5, 10⟩;
```

By virtue of the above declaration, b has the following properties:

(1) b is a row of 5 objects, each object being a row of 10 BOOLs.

(2) length(b) is 5.

(3) b can be subscripted, e.g., b[3], the result being a row of 10 BOOLs.

(4) length(b[i]) is 10 for i between 1 and 5.

(5) b[i] may be subscripted, e.g., b[i][j], the result being a BOOL.

In EL1, repeated subscripting is always written in the format $x[i_1][i_2]\ldots[i_n]$. Each application of a subscript to an object yields a new object whose order is one less. We assume that where the above notation is found awkward, syntactic extensions will be made to allow the more

familiar format $x[i_1, i_2, \ldots, i_n]$.

One final point should be made concerning notation. The operator ROW can be alternately denoted by the symbol R, allowing somewhat more compact definitions. For example, bool-matrix may be defined

```
bool_matrix ← R(R(BOOL));
```

with identical meaning.

3.9.3 STRUCT

A row is subject to the restriction that all its components have the same mode and identical sizes. A second class of modes, struct, allows composite objects whose components do not necessarily have the same mode. The builtin operator STRUCT takes as arguments a set of pairs each consisting of the name of the component and its mode. STRUCT delivers the mode which describes objects so constructed.

For example, the following definition might be used to represent a household fuse.

```
fuse ← STRUCT(amps : INT,
              manufacturer : R(10, CHAR),
              blown_flag : BOOL);
```

The mode fuse is the data type defined as follows. An object of mode fuse consists of three components:

(1) an INT

(2) a row of 10 CHARs

(3) a BOOL.

Since rows are homogenous objects, it is appropriate to select their components by numerical subscripts (e.g., x[i]). However, structs are

typically inhomogenous and it is useful to refer to their components by symbolic names. The above definition specifies both the modes of the components of a fuse and the names of these components. That is,

(1) the INT is named "amps",

(2) the row of 10 CHARs is named "manufacturer",

(3) the BOOL is named "blown-flag".

Fuse having been defined, it can be later used in declaring variables.

```
DECL kitchen_fuse, basement_fuse : fuse;
```

The variable kitchen-fuse is a fuse; hence, it has three components one of which is an INT named "amps". A component may be selected by name qualification which is denoted by the object name, followed by a period, followed by the name of the component. For example,

```
kitchen_fuse.amps
```

Since this is an INT, it may be given an INT value

```
kitchen_fuse.amps ← 15;
```

and used as an operand for an arithmetic expression

```
basement_fuse.amps ← kitchen_fuse.amps + 5;
```

We have discussed two methods of selecting a component of a compound object: subscripting (e.g., x[i]) and name qualification (e.g., kitchen-fuse.amps). These may be intermixed. For example, kitchen-fuse.manufacturer is of mode R(10, CHAR); hence, it may be subscripted, e.g., kitchen-fuse.manufacturer[i], yielding a CHAR. If we define

```
fuse_box ← R(4, fuse);
DECL central_control : fuse_box;
```

then we may write

```
central_control[2].blown_flag
```

obtaining a BOOL, or

```
central_control[i].manufacturer[1]
```

obtaining a CHAR. In general, selection proceeds from left to right, obtaining successively lower-level components.†

It is occasionally useful to compute which component of a struct is to be selected, as in the case of rows. This is denoted by subscripting. For example, consider

```
complex ← STRUCT(re:INT, im:INT);
  .
  .
  .
DECL z:complex;
```

The form z[1] has precisely the same meaning as z.re and z[2] is the same as z.im; z[i] is one of these two depending on the value of i.

In the discussion of rows, we introduced the notion of length unresolved modes resulting from forms such as ROW(INT) which defer binding of the number of components. Because structs may have components whose types are such modes, it is possible to have length unresolved modes of class struct. For example,

```
string ← ROW(CHAR);
shipment ← STRUCT(item:string, quantity:INT);
```

Since string is length unresolved, so is shipment. To declare a variable of

---

†The ability to intermix subscripting and name qualification and have this simple rule hold is the principal reason for the format chosen to denote name qualification (e.g., "kitchen-fuse.manufacturer"). If instead EL1 used the format used in Cobol or Algol 68 (e.g., "manufacturer of kitchen fuse"), a more complex reading algorithm would be required.

mode shipment, the unresolved dimension must be supplied, e.g.,

```
DECL j6 : shipment SIZE ⟨16⟩;
```

which specifies that j6.item has length 16. If several dimensions are unresolved they must all be supplied, for example, in the format $\langle d_1, d_2, \ldots, d_n \rangle$. The correspondence between the d_i's and the unresolved lengths in the struct is established by traversing the struct definition top-to-bottom, left-to-right (prefix walk) furnishing successive d_i's whenever an unresolved length is encountered. For example, consider

```
intρ ← R(INT);
string ← R(CHAR);
matrix ← R(R(BOOL));
matrixρ ← R(matrix);
comp ← STRUCT(a: intρ, b: matrixρ, c: string);
DECL x: comp SIZE⟨10, 3, 20, 25, 6⟩;
```

This results in the following:

(1) length (x.a) is 10

(2) length (x.b) is 3

(3) x.b[i] is a matrix, for i = 1, 2, 3

(4) length (x.b[i]) is 20, for i = 1, 2, 3

(5) length (x.b[i][j]) is 25, for appropriate i and j

(6) length (x.c) is 6.

As with ROW, it is useful to abbreviate the name STRUCT by its first letter. For example, comp could be defined

```
comp ← S(a: intρ, b: string, c: matrixρ);
```

3.9.4 PTR-ANY

One language feature that finds Algol 60, Lisp 1.5, and Algol 68 at odds is the existence and usage of pointers.† Algol 60 lacks the notion. Lisp 1.5 uses pointers almost exclusively and hence can consistently suppress the notion. Algol 68 allows pointers, generally making their appearance explicit.‡ In this respect, EL1 is most like Algol 68. However, it goes farther toward strict constructionism: a pointer will, on occasion, appear explicitly in EL1 where a corresponding pointer in Algol 68 will have its appearance suppressed.

In section 3.4, we mentioned the mode constant PTR-ANY but deferred explanation; we now remedy this omission. PTR-ANY denotes a primitive mode which can be intuitively described as: the set of objects which can point to other objects, with no restriction on the mode of the objects so pointed to. Consider, for example,

```
DECL p1, p2 : PTR_ANY;
```

The variables p1 and p2 can point to (reference) objects of any mode. Further, if p1 points to an object,

```
p2 ← p1
```

copies the value of p1 (essentially an address) into p2 so that p1 and p2 both point to the same object.

The question arises: how does one get p1 pointing to an object in the first place? The form

---

† By "pointer", we refer generically to data objects which contain the address of ("point to") other data objects.

‡ A fourth possibility is displayed in the treatment of pointers in PL/I [IBM66a] . Here, pointers exist and appear explicitly, but suffer a significant defect. It is possible, in fact easy, to inadvertently address an object of mode $\mathcal{M}_1$ and treat it as if it were of mode $\mathcal{M}_2$. Such errors are not detected.

```
p1 ← x
```

where x is some object, say an INT, will not work, for this would be interpreted as: copy the value of x (an INT) into p1 (a PTR-ANY) which would not achieve the desired result.† In general, there is no way to take a declared object x and obtain a pointer to it. This is neither an accident of design nor an arbitrary decision; the reason will be discussed in section 7.1.2.

It is, however, possible to create a new object which can be pointed to by a PTR-ANY. For example, consider

```
p1 ← allocate (bool_matrix, ⟨5, 10⟩)
```

The right-hand side of the assignment creates a new object of mode bool-matrix (dimension 5 by 10) and returns a pointer to the object; the assignment operation copies this pointer into p1. Hence, p1 points to the bool-matrix. This bool-matrix differs from all objects discussed thus far: it was created by an allocation, not a DECLaration, and therefore has no name. It can be designated only by means of pointers which contain its address. If p1 is assigned to p2

```
p2 ← p1
```

then both p1 and p2 point to the same bool-matrix.

The link between a pointer $\mathscr{P}$ and an object $\mathscr{O}$ to which it points is provided by a primitive function val, i.e., $\text{val}(\mathscr{P}) \equiv \mathscr{O}$. For example,

```
val(p1)
```

is the bool-matrix allocated earlier. In particular,

†In fact, mode checking performed on assignment will detect this as an illegal operation, for a PTR-ANY cannot contain an INT value.

```
val(p1)[5]
```

is the 5th component of the bool-matrix, a row of 10 BOOLs, and

```
val(p1)[5][3]
```

is a BOOL. Hence,

```
val(p1)[5][3] ← TRUE
```

is a legal form which sets the (3, 5) element of the matrix to TRUE.

As p1 is a PTR-ANY and hence unrestricted in the mode of objects it can address, it is legal to perform

```
p1 ← allocate(bool_matrix, ⟨5, 10⟩);
p2 ← p1;
p1 ← allocate(INT, ⟨ ⟩);
```

This leaves p2 pointing to the bool-matrix and p1 pointing to a single INT. In general, the function allocate takes two arguments: a mode (i.e., a mode-valued form) and a size specification; it returns a pointer to an object of that mode and size. In the case of INT, or any other length resolved mode, no size specification is required. The empty size specification is denoted by the form "⟨ ⟩" whose meaning will be discussed in section 3.11.

3.9.5 PTR

We have emphasized that objects of mode PTR-ANY are unrestricted in the mode of objects they can reference precisely because there are other pointers which are so restricted. We next consider this type of pointer.

The builtin operator PTR takes a mode $\mathfrak{M}$ as argument and produces the mode: the set of pointers restricted to address objects of type $\mathfrak{M}$.

For example,

```
intρ ← ROW(INT);
intρptr ← PTR(intρ);
DECL ip1, ip2 : intρptr;
```

The variable ip1 is of class pointer (abbreviated "ptr") but its specific mode is intρptr. It can point only to an intρ. Hence,

```
ip1 ← allocate(intρ, ⟨n⟩);
```

is legal, but

```
ip1 ← allocate(fuse_box, ⟨ ⟩);
```

is not. The right-hand side of the latter form returns an object of mode PTR(fuse-box); assignment of this to an intρptr is a type-error. As with PTR-ANY, it is possible for two objects of mode intρptr to refer to the same object, e.g.,

```
ip2 ← ip1
```

leaves ip1 and ip2 equal and val(ip1) is now identical to val(ip2).

We have thus far omitted discussion of pragmatics; however, a pragmatic note is unavoidable at this point lest it appear that pointer modes of restricted referent are an arbitrary whimsey. From considerations based only on semantic grounds, the charge is well-founded. The variable ip1 can be used for no purpose for which p1 (having mode PTR-ANY) could not be used; p1 can point to any object to which ip1 can point, the assignment p1 ← ip1 being legal. Modes such as intρptr are introduced for two reasons. (1) In many implementations it will be possible to use less storage for an intρptr than for a PTR-ANY since the latter will carry a type code as well as an address. (2) Possibly more important is that tightly bound modes

such as intρptr allow more efficient (i.e., complete) compilation. For example, the compiler when confronted with the form

val(ip1)[k]

can determine that it is well-formed, that it involves subscripting a row of INTs, and that it yields an INT. Code generation is straightforward. The analogous form involving a PTR-ANY

val(p1)[k]

could have any number of possible selection operations and result types, depending on what sort of object p1 references at the time the form is evaluated. Code compiled for this must reflect the uncertainty.

It is frequently useful to deal with pointers which can reference objects of more than one mode† but which are not totally unrestricted. The mode of such a pointer is said to be of sub-class united pointer. For example,

int_or_bool_ptr ← PTR(INT, BOOL)

The operator PTR can be given more than one argument and when so invoked it produces a mode defining pointers which may reference more than one type of object. If we declare

DECL q : int_or_bool_ptr;

then q can point to INTs or BOOLs but to nothing else.

PTR-ANYs and united pointers can, at different times in the course of a program, reference objects of different modes, possibly in a fashion not known when the program was written. Hence, it is sometimes useful to

---

†For example, a Lisp 1.5 pointer can reference an atom, a two-word block (cons cell), a fixed point number, or a floating point number.

determine the mode of the object referenced by a given pointer. The builtin function mval has been provided for this purpose. Mval, applied to any value of class ptr yields the mode of the object pointed to. Hence, if

```
p1 ← q ← allocate(INT, ⟨ ⟩)
```

then mval(q) is INT as is mval(p1).

3.9.6 Summary

Section 3.9 has discussed four† groups of mode-valued forms: constants, rows, structs, and pointers. Since a large number of notions have been presented in a loose, discursive exposition, a table summarizing this material may be useful.

Group	Sub-Group	Example
constants	non-pointer	INT
	pointer	PTR-ANY
row	resolved	ROW(5, CHAR)
	unresolved	ROW(INT)
struct	resolved	S(a:INT, b:R(5, CHAR))
	unresolved	S(a:INT, b:R(CHAR))
pointer	simple	PTR(ROW(INT))
	united pointer	PTR(ROW(INT), BOOL)

†A fifth group, rany, can best be presented after discussion of procedures. It will be treated in section 3.14.2.

3.10 SELECTION

The operation of selection has been discussed as a peripheral issue in sections 3.9.2-5. In this section, we summarize this discussion and treat one additional point.

The operators ROW and STRUCT produce modes of class row and struct, respectively; objects whose modes belong to these classes are said to be rows and structs. Such objects are compound and consist of components which may be designated by selection operations. There are two such operations: qualified naming and subscripting. The former is applicable only to structs. It is denoted by writing the object, followed by a period, followed by the name of the component (e.g., z.re), the name of the component having been specified in the mode definition. Subscripting is applicable to either rows or structs. It is denoted by writing the object, followed by a form enclosed in square brackets (e.g., x[i+j]). The operation is carried out by evaluating the bracketed form which is expected to yield an INT result, say i, and taking as result the i^{th} component of the object.

The object on which selection is performed is not restricted to be an identifier: any form whose value is a row or struct may be employed. For example, it can be a prior selection (e.g., a[i].re), in which case the selection is carried out from left to right. It can be a function which delivers a row or struct (e.g., f(x)[i]). In particular, it can be the application of val to a pointer; e.g., if p points to a bool-matrix, then

```
val(p)[i][j]
```

is a legal form having a BOOL value.

In connection with the last example, one additional point bears mentioning. Since such forms occur frequently, it is desirable to abbreviate their

representation. An abbreviation is immediate if it is observed that the appearance of "val" is redundant. Given that p is a pointer, it is clear that p cannot itself be the object of a selection, for it has no components. Hence,

```
val(p)[i][j]
```

can be unambiguously abbreviated

```
p[i][j]
```

The latter is defined to be identical to the former. This schema of abbreviation is, of course, completely general. For example, if f(x) returns a pointer to a row of structs each of which has a component named "tx" which is a pointer to a bool-matrix, then

```
f(x)[n] . tx[i][j]
```

may be written with the same meaning as the explicit form

```
val(val(f(x))[n] . tx)[i][j]
```

3.11 AGGREGATES

In section 3.9 we used, with only marginal explanation, forms such as "⟨10⟩", "⟨10, 6, 3, 20, 25⟩", and "⟨ ⟩". We now turn to a systematic treatment of such forms which are termed aggregates.

An aggregate is a special form used for creating compound values, i.e., rows and structs. Recall that we earlier defined the mode intρ by

```
intρ ← ROW(INT);
```

The simplest aggregates are of mode intρ. For example,

⟨10⟩ is an intρ of length 1

⟨5, 2+1, i, 3∗j⟩ is an intρ of length 4

⟨ ⟩ is an intρ of length 0

An aggregate of mode intρ is written by enclosing a sequence of forms, separated by commas, in angle brackets. The value of such an aggregate is an intρ whose length is the number of forms present.† The value of the i^{th} component of the intρ is the value of the i^{th} form. Hence,

```
⟨ 13, f(w, t) + 2 * f(x), [[ p(x) ⇒ 6 ; 3 * n ]]⟩
```

is a legal aggregate whose value is an intρ of length 3.

An aggregate whose value is other than an intρ is written by prefixing the list of components by the desired mode followed by a colon. For example, recall that the mode complex was defined by

```
complex ← S(re: INT, im: INT);
```

Assuming that x and y are INTs, an aggregate whose value is a complex may be written

```
⟨ complex : x, y⟩
```

The components of such an aggregate can, of course, involve computation

```
⟨ complex : (b * x) + c, z.re - z.im⟩
```

In general, an aggregate can produce a value having multiple sub-levels. For example, consider

```
triple ← ROW(3, INT);
initials ← ROW(3, CHAR);
record ← S(point1: triple, point2: triple, observer: initials);
```

Assuming that i, j, and k are ints, we can later write

---

†In general, the size specification in a declaration (c.f. §3.9.2) is an intρ. An aggregate, as above, is one convenient way of denoting such an intρ. However, any other form whose value is an intρ will do as well.

```
DECL t : triple;
t ← ⟨i, j, k⟩;
⟨record: t, ⟨triple: i-j, i-k, j-k⟩, ⟨initials: 'n, 'm, 'p⟩⟩
```

The value of the last line is a record. Syntactically, it is an aggregate containing three forms: the first is an identifier, the second and third are themselves aggregates.

In all the aggregates discussed thus far, the mode has been either resolved (e.g., triple) or length unresolved with dimension one. In the latter case, the unresolved dimension is deduced from the number of forms present. However, when the mode is unresolved with dimension greater than one, it becomes awkward to perform such deductions. Hence, we require the occurrence of an explicit SIZE specification, using the same format as in DECLarations. For example, consider the mode

```
free_record ← S(point1: intρ, point2: intρ, observer: string)
```

in which "points" have unresolved length and the "observer" component is a string (a sequence of CHARs of unresolved length). An aggregate producing a free-record may be written

```
⟨free_record : SIZE⟨3,4,6⟩ :
        ⟨x,y,z⟩,
        ⟨x-t, y-t, z-t, x+y+z⟩,
        ⟨string : 'w, 'a, 't, 's, 'o, 'n⟩⟩
```

The component point1 is an intρ of length 3, point2 is an intρ of length 4, and observer is a string of length 6.

3.12 PROCEDURES – DEFINITION AND APPLICATION

A procedure is an EL1 form whose value may depend on some number of arguments. In general, a procedure may have both side effects and a returned value. Hence, the notion of procedure call in EL1 embraces both the ⟨function call⟩ and ⟨procedure statement⟩ of Algol 60.

A procedure call may be denoted† by writing the procedure name followed by a list of arguments enclosed in parentheses, e.g.,

```
transform(38, u, z.im)
```

Each argument is a form; hence, assignments, compound-forms, mode-valued forms, and aggregates are acceptable arguments, e.g.,

```
transform(z ← x+y, ⟨i, j, k⟩, [[ p(q) ⇒ z.re ; f(z.im) ]])
```

The procedure name may be replaced by any form which evaluates to the desired procedure. For example,

```
[[ i > j ⇒ transform ; revert ]](38, u, z.im)
```

applies either transform or revert depending on whether or not i exceeds j.

Returning to the familiar case, a procedure name is a variable whose value is a procedure. Such variables are of mode proc-var and may be created by declaration

```
DECL foo, transform, revert : proc_var ;
```

---

†As discussed in section 3.6, the application of a procedure p to a single argument x may be denoted by "p ∘ x" instead of "p(x)". In general, any form satisfying the schema

⟨form1⟩ ∘ ⟨form2⟩

is completely equivalent to

⟨form1⟩ (⟨form2⟩)

Hence, only the latter format will be discussed in this section.

Like other variables, they may be given values of appropriate mode by assignment. One source of such values is proc-vars which already have values. For example, if "sign" is a proc-var which has a procedure value, then

foo ← sign

assigns the procedure value to foo. Hence, "foo(x)" is identical to "sign(x)".

Assignment of proc-var to proc-var merely distributes existing procedures. We now turn to the creation of new procedures. Consider, for example,

PROC(d:INT, n:INT) BOOL; n = d*(n/d) ENDP (4, 12)

This is a procedure application composed of an explicit procedure followed by a parenthesized list of arguments. An explicit procedure (or expr) consists of

(1) an opening bracket: PROC,

(2) a parenthesized list of formal parameters,

(3) a type which declares the result type, i.e., the mode of the result delivered by the expr,

(4) a procedure body,

(5) a closing bracket: ENDP.

The above expr takes two INT arguments and returns a BOOL value: TRUE iff the first argument is a factor of the second.

This is a particularly simple expr, for its body consists of the single form n = d*(n/d). In general, a procedure body consists of one or more declarations followed by a sequence of statements (c.f. §3.7), all separated by semicolons. For example, the above procedure application could be written

```
PROC(d:INT, n:INT) BOOL;
DECL r : INT;
r ← n/d;
n = d*r ⇒ TRUE;
FALSE;
ENDP (4, 12)
```

Here, the procedure body consists of one declaration and three statements.

The application of this procedure, and of procedures in general, is carried out as follows.

(1) The formal parameters are bound to the corresponding arguments. As this is a complicated process, we postpone discussion of the general case. In the above example, this results in the creation of two INT variables, d and n, and the assignments d ← 4 and n ← 12.

(2) The declarations are evaluated. In the above example, this results in the creation of an INT variable r.

(3) The statement sequence is evaluated as a compound form (c.f. §3.7). That is, the group of statements is evaluated as if it were a single compound form enclosed in the brackets BEGIN, END. The value of the compound form is the basic value of the procedure.

(4) The mode of the basic value is checked against the declared result type. With a few exceptions to be discussed in the formal definition, these must be the same and, if so, the basic value is the value of the procedure.

While it is possible to apply an expr directly to its arguments, this is not commonly done. Instead, the value of the expr is assigned as the value of a proc-var; subsequently the proc-var is used to denote the procedure.

For example,

```
DECL factor : proc_var;
factor ← PROC(d:INT, n:INT) BOOL; n = d*(n/d) ENDP;
factor (4, 12)
```

At this point, it may be useful to examine a non-trivial procedure which displays the use of the various forms discussed thus far. We consider a procedure which takes two INT arguments and returns TRUE if and only if there exists a perfect† number between them.

```
test_for_perfect ←
PROC (lower:INT, upper:INT) BOOL;
DECL sum : INT;
DECL flag : BOOL;
flag ← FALSE;
FOR k ← lower, ..., upper TILL flag DO
    BEGIN
    sum ← 0;
    FOR j ← 1, ..., k-1 DO
        [[ factor(j, k) ⇒ sum ← sum + j ]];
    sum = k ⇒ flag ← TRUE;
    END;
flag ENDP;
```

This tests each successive integer k between lower and upper comparing the sum of its factors with k. If sum = k then flag is set TRUE and the

---

†A number n is said to be perfect if it equals the sum of its factors (including 1 but excluding n). For example, 28 has factors 1, 2, 4, 7, 14 which sum to 28; hence, 28 is perfect.

testing stops. The value of the procedure is the final value of flag.

It should be noted that the variables j and k are not declared. In general, the index variable of an iteration form is not declared and is local to its form, having no existence outside that form. If, in the above procedure, there were a declaration

```
DECL k : INT ;
```

the k so declared would be superceded within the iteration by the index-variable k.

In specifying the process whereby a procedure is evaluated, we postponed discussion of formal parameter binding. We now consider this process. In the most general case, a formal parameter consists of

(1) a name, e.g., "x",

(2) a type, e.g., "INT", which is either an identifier or a mode-valued constant,

(3) a bind-class.

The bind-class may be omitted, as has been the case in the examples given thus far, or it may be any of the three symbols BYVALUE, BYREF, or UNEVALED. An omitted bind-class is given the default value BYREF. Each bind-class has a distinct method of binding.

Binding BYVALUE is relatively straightforward. First, the argument is evaluated. Then a new variable is created, with name and mode as specified in the formal parameter. Finally, the value of the argument is assigned to the created variable. Hence, binding BYVALUE involves little more than the combination of three notions previously discussed: evaluation of a form, creation of a new variable, and assignment. There is, however, one complication. Consider a formal parameter

x : intρ BYVALUE

with corresponding argument

⟨5, 19, 26, 83, 37, 12⟩

The latter evaluates to an intρ $\mathscr{V}$ of length 6. If x were created as a declared variable, its declaration would necessarily include a size specification, e.g.,

DECL x : intρ SIZE ⟨6⟩;

Since x is instead a formal parameter which is to assume $\mathscr{V}$ as initial value, the SIZE specification of x is derived from the size of $\mathscr{V}$. We say that the mode of the formal parameter x is length resolved by the size of its argument (just as the mode of the declared variable x is length resolved by the SIZE specification which follows it).

To discuss binding BYREF, it is necessary to first introduce a dichotomy which has been suppressed thus far. The term "value" has been used in a sense which embraces two distinct classes of data: proper objects and pure values. Proper objects include declared variables, formal parameters, objects created by allocate, and the components of proper objects. In terms of implementation, a proper object is a data object which is stored in some identifiable place. A pure value, in contrast, has no place where it can be said to reside. Pure values include the values of constants, the values of aggregates, the results of binary operators such as =, ≠, +, -, and *, as well as the results of some procedures.

To perform a binding which is specified BYREF the corresponding argument is first evaluated, yielding a value $\mathscr{V}$. If $\mathscr{V}$ is a pure value, then the

specification BYREF is ignored, and the binding is carried out BYVALUE.† If, however, $\mathscr{Y}$ is a proper object the BYREF binding is carried out as follows. Consider for example the formal parameter

x : INT BYREF

with argument y which has mode INT. No new data object is created for x; instead, the binding arranges that inside the procedure, x names the same object as y names at the place of call. In particular, if an assignment such as

x ← 39

is evaluated in the procedure, then y is given the value 39. Hence, the procedure may have side-effects caused by assignments to the formal x.

There is no restriction that the argument be an identifier in order for BYREF binding to be carried out. For example, if b is an intρ defined at the place of call then a legal argument, of class proper object, is

b[i+3*j]

Similarly, if z is complex then the argument

z . re

would evaluate to a proper object. Finally, if p is a PTR(INT) then a proper object is obtained from the argument

val(p)

---

†This apparent anomaly is necessitated because the binding class BYREF is incompatible with a pure value argument. BYVALUE binding is taken as a plausible fall-back position which in most cases achieves the desired result. The system could instead deem this an error, but such a dictum would, in general, be found too austere.

Binding a formal parameter UNEVALED differs in one significant way from binding either BYVALUE or BYREF. In the latter cases, the first step in binding is to evaluate the argument which, according to the syntax, must be a form. Binding UNEVALED means that the form is not evaluated: the formal parameter is bound to the unevaluated argument. The formal parameter must be of mode form. For example, consider the formal parameter

x : form UNEVALED

with argument

f(z) + p.a

After binding, x is an object with mode form and has as value the form "f(z) + p.a". If the procedure is to do anything interesting with x, there must be operators and procedures which take arguments of mode form. One such procedure is eval which takes a single form and delivers its value. Eval and other such procedures will be discussed in section 5. Here we note that a common use of UNEVALED arguments is in the construction of procedures which evaluate certain arguments conditionally or only after certain parts of the procedure body have been executed.

The final step of procedure evaluation is termed proc-exit. This takes the value $\mathscr{V}$ of the statement list and produces the value $\mathscr{V}'$ of the procedure. With some exceptions to be discussed in subsequent sections, $\mathscr{V}$ and $\mathscr{V}'$ are generally equal but they need not be the same object. There are two primary cases.

(1) $\mathscr{V}'$ will be a pure value if either (a) $\mathscr{V}$ is a pure value or if (b) $\mathscr{V}$ is a proper object whose scope is the procedure being exited (e.g., if $\mathscr{V}$ is a variable declared in that procedure). In sub-case (b), $\mathscr{V}'$ is a copy of the object $\mathscr{V}$. The latter must be destroyed during proc-exit.

(2) If neither (a) nor (b) holds then $\mathscr{V}'$ will be a proper object, the same object as $\mathscr{V}$.

The significance of this distinction will become apparent in the next section.

We have now considered each of the forms defined in EL1 and have thereby concluded the first phase of the informal description. We now turn to a number of advanced topics which have been ignored in the interest of initial simplicity.

3.13 LEFT-HAND VALUES

The notion of left-hand values or L-values arose from attempts to give precise explanation of assignment and related operations such as procedure binding. The term "L-value" was coined by the designers of CPL [CPL66] and it will be useful to establish terminology by examining how the notion is treated in that language. For example, consider the CPL assignment†

```
(if p(x) then i else j) := k
```

which sets either i or j to the value of k, depending on whether or not p(x) is true. The CPL explication of this is as follows.

(1) The left-hand side is evaluated in L-mode††, producing an L-value.

(2) The right-hand side is evaluated in R-mode, producing an R-value.

(3) The R-value obtained in step 2 becomes associated, by virtue of the assignment, with the L-value obtained in step 1.

---

†To make the meaning of the construct clear, we have taken liberty with the syntax of CPL and rendered the fragment in pseudo-Algol. In proper CPL, the above line would read

```
(p(x) → i, j) := k
```

††CPL here uses the term "mode" in the sense of "manner". There is no connection with the EL1 use of "mode" meaning "data type".

The essential point is that in CPL the normal mechanism (i.e., R-mode) for evaluating forms such as "(if p(x) then i else j)" produces a result which would make the assignment meaningless. L-mode evaluation operates in such fashion that the result it yields is an appropriate candidate for being assigned to.

Speaking in implementation terms, an L-value is a storage location and an R-value is a bit pattern. An R-value $\mathfrak{R}$ is associated with an L-value $\mathfrak{L}$ by setting the contents of location $\mathfrak{L}$ to be $\mathfrak{R}$. An expression evaluated in R-mode produces a bit pattern (e.g., held in the accumulator); an expression evaluated in L-mode produces a storage location. Were the notion of pointer present[†] in CPL, the following would define the relation of R-value and L-value.

> The R-value of a pointer p is the L-value of the object to which p points.[‡]

Using this terminology, we can discuss the treatment of values in EL1. Unlike CPL, and almost all other languages, EL1 uses the convention that all forms are evaluated in L-mode and all values are L-values. We call this the locative condition. In implementation terms, whenever a form is evaluated the actual result is a storage location. If, for example, a form has INT value, the evaluator obtains the address of a storage location containing a bit pattern which is to be interpreted as an integer (e.g., a 36 bit quantity in two's complement representation). In an assignment such as

```
x ← y
```

the evaluator obtains two addresses and then copies the contents of the

---

[†] Its absence is a real weakness of the language; however, the above relation should be clear even in its absence.

[‡] If this is unclear, the following diagram may be helpful:

right-hand address into the left-hand address. In consequence, the following forms are all legal in EL1:

```
[[ p(x) ⇒ i; j ]] ← k
val(r) ← ⟨x, y⟩
val(r).re ← z.im
f(z) ← f(w)
```

The last form deserves special comment. If the value of f(z) is a pure value then assigning to it the value of f(w) accomplishes no useful purpose, for this value is lost once the assignment has been performed. If, however, the value of f(z) is a proper object then assignment to it is an assignment to some variable or allocated object. For example, suppose p is a PTR(intρ) such that val(p) has value ⟨2, 4, 6, 8⟩. Further, suppose f is defined

```
f ← PROC(x: intρ BYREF, i: INT) INT;
      x [length(x) - i] ENDP;
```

Consider

```
f(val(p), 1) ← 100
```

The value of the left-hand side is the third component of the intρ to which p points. The assignment changes this to have value 100. Hence, val(p) now has value ⟨2, 4, 100, 8⟩.

3.14 ADDITIONAL TOPICS CONCERNED WITH MODES

Several aspects of mode handling in EL1 have been postponed because their in-line presentation would have required extensive forward reference. We now address them.

3.14.1 Initial Values

The first of these is relatively minor and requires mention only because it goes against tradition: all objects in EL1 are given an initial value. Unlike Algol 60, in which a declared variable is said to be "undefined" until an assignment to it has been executed, EL1 explicitly rules out the notion of an object with undefined value. Formal parameters obtain initial values from their arguments; declared variables and allocated objects are initialized by the evaluator to default values based on their data type. For each of the following four primitive modes there is a default value:

INT	has default value 0
BOOL	has default value FALSE
CHAR	has default value the blank character
PTR-ANY	has default value NIL

The default value of an object of mode $\mathfrak{M}$ is determined as follows.

(1) If $\mathfrak{M}$ is in the above list, the value is as specified.

(2) If $\mathfrak{M}$ is of class ptr, the value is NIL (c.f. §3.16.1).

(3) If $\mathfrak{M}$ is of class row or struct, each component is given the value obtained by applying this procedure to the mode of that component.

Since all objects are initialized, it is legal to use an object without having explicitly given it a value, e.g.,

```
DECL x : INT;
DECL b : BOOL;
foo(x, b)
```

The values of the arguments to foo are 0 and FALSE. Since these values are defined, rather than an accident of the implementation, they may be used without fear that a different implementation will produce different results.

3.14.2 Generic Procedures and the Operator RANY

There is one additional builtin operator which produces modes: RANY, which stands for "restricted any". The notion of RANY is motivated by the observation that certain conceptual operations are applicable to objects of several different data types. For example, the notion of norm is well-defined on INTs (meaning absolute value), complex numbers (meaning $\lceil\sqrt{x^2+y^2}\rceil$) and strings (meaning the number of CHARs in the string).† In such circumstances, it is desirable to denote the operation by a single procedure name which is applicable to arguments of several modes, e.g.,

norm(i+j), norm(⟨complex: x, y⟩), norm(stringvar)

The procedure invoked by "norm" must accept an argument of several possible modes: INT, complex, or string. In such circumstances, the mode of the corresponding formal parameter is said to be of class rany. Consider the definitions

```
DECL item : mode;
DECL norm : proc_var;
item ← RANY(INT, complex, string);
norm ← PROC(x: item) INT;
        typ(x) = INT ⇒ sign(x) * x;
        typ(x) = complex ⇒ sqrt((x.re * x.re) + (x.im * x.im));
        typ(x) = string ⇒ length(x);
        ENDP;
```

---

†At the cost of complicating our example, we could add intρ, bool-matrix, and a host of other modes to this list.

The definition of item may be interpreted roughly as: item is the mode consisting of INTs or complexes or strings. We say that INT, complex, and string are the alternatives of item, that item is of class rany, and that item is type unresolved.

The reasons for this choice of terminology will become clear if we consider the evaluation of a form such as

norm(f(a))

which proceeds as follows.

(1) The argument f(a) is evaluated, producing a value with some mode $\mathfrak{M}$.

(2) Since the formal parameter x is declared to have mode item and since item is type unresolved, the mode $\mathfrak{M}$ of the argument is used to resolve it. If $\mathfrak{M}$ is one of the alternatives of item then the binding is legal and $\mathfrak{M}$ becomes the actual mode of x; otherwise, the binding is illegal and an error results.

(3) The binding of x and the evaluation of the procedure occur just as if x had been declared to have mode $\mathfrak{M}$.

In the procedure body, x is either an INT or a complex or a string; the choice having been made on procedure entry, the mode of x cannot change in the course of a given procedure activation.† Since each case requires separate handling, it must be possible to determine in the body of norm which of the three cases has occurred. To allow this and related testing

---

†It is here that the difference arises between rany in EL1 and union in Algol 68. If EL1 allowed union and if x had been declared as a formal parameter of mode

UNION(INT, complex, string)

then x would assume one of these modes on procedure entry, but would be free to change to any of these modes during that activation.

there is a builtin function typ which, applied to any value, yields the mode of that value. For example, the second line of norm reads: if typ(x), the actual mode of x on this activation of norm, is INT then multiply x by its sign and return this value.

The process of type resolving a rany is closely related to that of length resolving a row. In each case, there is some commitment that has not been made at the time a procedure is written: the length of some object in the case of rows, the choice of one mode among several in the case of ranys. The commitment is postponed to the time of procedure activation, at which time an attribute of the argument resolves the uncertainty.

The use of ranys is not restricted to formal parameters. It is occasionally useful to specify that a declared variable takes its mode from a finite set, deferring choice until run time. For example, in

```
DECL z : item SPECIF p(n);
```

z has declared mode item. The actual mode is determined by p(n) when the declaration is evaluated. The value of p(n) should be a mode in the set {INT, complex, string}. If so, this value becomes the actual mode of z; if not, the specification is illegal and an error results.

3.15 MODE RECURSION AND FORWARD REFERENCE

One of the subtle problems which arises in providing a data type definition facility is mode recursion and forward reference. In this section, we present the problem, outline its solution in other languages, and discuss how it is handled in EL1.

As an example of the pitfalls readily accessible, consider the definition set (in some language, not necessarily EL1)

```
DECL m1, m2 : mode;
m1 ← STRUCT(a:INT, b:m2);
m2 ← STRUCT(c:CHAR, d:m1);
```

The intention is to define m1 (and m2) as being recursive; if x is of mode m1 then x contains a proper sub-part, denoted "x.b.d", which is also of mode m1. Such a property is clearly unusual and perhaps disturbing.† One might ask whether any meaning can be attached to such definitions.

The answer to this depends on the model chosen for storage allocation. Until now, our discussion has been free from consideration of such issues for the elementary notions in data type definition are independent of implementation. Here, however, the axioms we choose for modes depend critically on our model. If an instance x of a mode $\mathfrak{M}$ is regarded as filling some fixed segment of core, then for x to contain as proper sub-part an object of mode $\mathfrak{M}$ is impossible. If, however, no such requirement is imposed then implementation (and subsequent rationalization) present only minor difficulty.††

For example, the policy might be adopted that any composite object of n components is to be implemented by a vector of n pointers ("pointers" in the systems programming, not EL1, sense). Creating an instance x of mode m1 is then carried out by allocating a two-pointer vector and initializing the first pointer to reference a separately allocated INT. If "x.b" is

---

†For example, Morris [Mor68] in his study of a type system in lambda-calculus models of programming languages explicitly rules out type expressions having this property.

††So that it is clear we are not merely raising a strawman, we wish to emphasize that recursive modes have appeared in proposals for extensible languages, for example the data definition facility of Standish [Stand67].

ever evaluated, an object of type m2 is allocated and field "b" of x is set to reference it. Uses of deeper components of x (e.g., "x.b.d.b") are handled analogously: x grows as references to deeper levels are made, storage allocation being driven by the evaluation.

The difficulty in handling recursive modes is not confined to the allocation of storage for mode instances. Processing the mode definition is also made more difficult for the usual problem of binding the defined name in a recursive definition arises.† Hitherto, we have regarded a mode such as

---

† For example, consider a definition of factorial

factorial(n) = if n = 0 then 1 else factorial (n-1)

Suppose we wish to convert this to a definition in which the variable "factorial" is bound to an equivalent λ-expression. We might try

factorial = λn. if n = 0 then 1 else factorial (n-1)

However, this would be incorrect, for the variable "factorial" used on the right-hand side is a free variable of the form whereas we want to identify it with the value of the defining form. That is, the desired value for "factorial" on the right-hand side is the one what will be established by the definition. This is the typical manifestation of any recursive definition. To get the desired result, we proceed as follows. Rewrite the definition as

factorial = {λf. λn. if n = 0 then 1 else f(n-1)} factorial

this has the format

A = F A

where

F = {λf. λn. if n = 0 then 1 else f(n-1)}

For the λ-calculus, there is a fixed point operator Y (or paradoxical combinator, c.f. [Cur58]) having the property that for any well-formed formula F,

YF = F(YF)

Using F, we can obtain a solution to A = FA, namely

A = YF

since

A = YF = F(YF) = F

Hence a correct definition of factorial, from which the circularity has been removed is

factorial = Y {λf. λn. if n = 0 then 1 else f(n-1)}

```
m1 ← STRUCT(a:INT, b:m2);
```

as an assignment in which the left-hand side derives value from an operator acting on the operands in the right-hand side. In the case of

```
m1 ← S(a:INT, b:m2);
m2 ← S(c:CHAR, d:m1);
```

this schema fails. The value of m2 required in the first line is not the one defined at that point but rather the value established in the second line. That is, the mode recursion manifests itself as a logical circularity in the definition. Formally, the required definition may be rendered in the λ-calculus as

$$(m1, m2) = Y\{\lambda(x, y).\ (S(a:INT,\ b:y),\ S(c:CHAR,\ d:x))\}$$

where Y is a fixed point operator.†

The desired definition can be obtained, but only by some mechanism more complicated than simple assignment. One possibility is to implement an operator Y which handles modes in our programming language (for a related example, c.f. [Land66c]) and convert the above definition into an

---

† This is a purely formal definition written in analogy with recursive procedure definitions such as that for factorial. To derive it, we observe that m1 and m2 must satisfy the relations

$$(m1, m2) \equiv (STRUCT(a:INT,\ b:m2),\ STRUCT(c:CHAR, d:m1))$$

This may be rewritten as

$$(m1, m2) = \{\lambda(x, y).\ (S(a:INT,\ b:y),\ S(c:CHAR,\ d:x))\}(m1, m2).$$

If Y is a fixed point operator having the property that $YF = F(YF)$, then using the same argument as given for factorial, we obtain

$$(m1, m2) = Y\{\lambda(x, y).\ (S(a:INT,\ b:y),\ S(c:CHAR,\ d:x))\}$$

assignment to the pair (m1, m2). While in principle this can be made to work, there does not appear to be any simple or efficient technique. Hence, the introduction of Y would violate the dictum of parsimony as well as economy.

Another possibility is to imitate the Algol 60 procedure declarations and require such mode definitions to be recast into declarations which are treated specially by the evaluator. For example, in pseudo-Algol, this would read

```
begin
    mode m1 struct(a: int, b: m2);
    mode m2 struct(c: char, d: m1);
```

Special treatment includes recognizing the mutual recursion and handling it in a fashion analogous to the mutual recursion of procedures.

A third solution is given by Standish's data definition facility [Stand67] which allows the definition†

```
m1 ← S(a: INT, b: m2);
```

to remain an executable assignment statement, but recognizes m2 as "undefined". For each such variable which is used before its definition, a use chain is constructed consisting of all places in which it is used. When m2 is defined

```
m2 ← S(c: CHAR, d: m1);
```

the use chain is searched and the definition of m1 is reexamined and

---

† We have taken our usual liberty to change notation. Standish would write

```
m1 ← [a: int | b: m2];
```

completed. It should be noted that this scheme requires some additional mechanism to handle the definition of m2 which depends on the partially defined variable m1.

Having explored this simple example of mode recursion in some detail, we turn to another class of problem cases. Consider the definition

```
m3 ← S(a:INT, b:PTR(m3));
```

The mode m3 is not recursive, for an instance of m3 does not contain an m3 as a proper sub-part; the component "b" merely points to an m3. The definition does, however, use m3 on its right-hand side before m3 is defined and hence is said to involve a forward reference. Note that here, as with recursive modes, use of forward reference is an intrinsic attribute of the mode, not an accidental property of this particular definition. By logical extension we speak of the mode as being of forward reference. Clearly, recursive modes are a subcase of forward reference modes. Also, it is clear that some of the difficulties in processing the former occur with all forward reference modes.

The set of problem cases does not stop here. Were we to consider definitions allowing unions (in the Algol 68 sense, c.f. §2.2.2), a plethora of additional difficulties arise.† We will avoid all these and summarize the two issues raised above.

(1) Should recursive modes be permitted? If so, how should they be expressed in the language and how should they be handled by the evaluator?

(2) Should other modes of forward reference be permitted? If so, how expressed and how handled?

There are, of course, no absolute answers to such questions. Each language must strike its own balance in trading off generality for the efficiency of special cases. One can conceive of several well-designed languages which answer these questions in radically different ways. In the case of EL1, there are several relevant design constraints which largely determine its answers.

To begin with, the most important of these, it will be recalled that EL1 is to serve as the base for an extensible language. Hence, those language features which can be reasonably defined in terms of more primitive features should be excluded from EL1 proper and left for extensions. As outlined above, objects having recursive modes can be constructed using pointers, so that recursive modes can be obtained as an extension. Indeed, were recursive modes built into the language, they doubtlessly would be implemented in a similar fashion. Therefore, it is not clear that a significant advantage is obtained by putting recursive modes into the base.

---

†Note that rany does not enter into this discussion as there are no objects of class rany and modes of class rany cannot be used in defining new modes.

Further, for a language to serve as an extensible language base it should be not only parsimonious but literal as well. That is, it should display its structure and mechanisms as clearly as possible, shunning semantic sugar. In particular, since pointers are a fundamental notion in EL1, it seems advisable to exhibit their appearance wherever possible. Hence, to use pointers in implementing recursive modes but suppress their existence seems unwise. (These considerations apply only to a base language. Such suppression is precisely that which is desired in packaging extensions offered to the public domain.) For these reasons, we do not admit recursive modes.

Turning to other modes of forward reference, the issue is not "whether?" but "how?". In any language with typed pointers, the construction must be admitted. Indeed, the basic construct for creating linked lists and similar structures is blocks of type $\mathfrak{M}$ containing one or more components which may point to other blocks of type $\mathfrak{M}$. In EL1, the issue is still more decisive, for certain data structures required by its evaluator necessitate forward reference modes.† Since the evaluator is an EL1 program, its data structures must be definable by the data type definition facility.

The implementation of forward reference modes is far less clear-cut. It is possible, of course, to use any of the techniques for handling recursive modes discussed earlier, with appropriate simplifications. For example, Basel [Jorr69] which allows forward reference but not recursion requires all operations of mode creation to be definitions occurring in the declaration portion of blockheads. All mode definitions in a block are processed together, using a three-pass scheme.

---

†That is, recursion in the concrete syntax requires use of forward reference in the abstract syntax. (c.f. §4.1 for an explanation of these terms.)

One disadvantage of this class of techniques is that they generally[†] involve restricting the right-hand side of mode definitions to a fixed set of mode operations (e.g., struct, row, ref, etc.) taking mode identifiers as arguments, with the single combination rule being functional composition. That is, the following type of construct is generally forbidden as a defining form

$$[\![\, p(x) \Rightarrow \text{complex};\ q(x) \Rightarrow R(4, \text{INT});\ R(\text{INT}) \,]\!]$$

The prohibition stems from the difficulty of handling arbitrary forms involving forward reference. While in principle one could use a special evaluator which leaves a hole, to be filled in later, for each forward reference, this is complex and expensive. By restricting the forms which can appear to an essentially fixed set, the processor is simplified at the expense of the language.

In the design of EL1, it was decided that all constructs should be treated as executable forms whenever possible. In particular, it was decided that mode definitions should be computable, so that the above construct should be legal. This required a radically different technique for handling forward reference modes, which we discuss below. Before doing so, it is necessary to discuss the definition of modes in greater detail than was possible in section 3.9.

There, the term "mode" was informally defined as the EL1 construct corresponding to the intuitive notion of data type. Formally, this statement needs refinement. Corresponding to each unique data type (whether built into the language or programmer-defined) is a compound object which holds needed information concerning this data type. This information includes:

[†] For example, Basel and Algol 68 both impose this restriction.

(1) functions for assignment and accessing components (in the case of compound objects), (2) a class code (row, struct, or ptr), and (3) a descriptor specifying components (in the case of compound objects). The object holding this information is said to be a data type definition block or ddb; i.e., the mode of this object is ddb. In other words, a ddb is a structure consisting of an assignment function, a selection function, a class code, a descriptor, and a few other items; for each data type there is a ddb which defines its properties.

Having defined the notion of ddb, we can give a precise definition of mode: a mode is a pointer to a ddb. That is, the builtin definition of mode is equivalent to

```
mode ← PTR(ddb);
```

Several consequences of this should be noted.

(1) Assignment of one mode to another entails copying a pointer, e.g., if m1 is a mode, then

```
m1 ← intρ
```

leaves m1 pointing to the same ddb that intρ points to.

(2) The value of a mode-valued constant is a pointer to a constant ddb. For example, the value of the constant INT is a pointer to the ddb that defines the primitive notion of integer in EL1. The assignment

```
m1 ← INT
```

sets m1 to point to this ddb.

(3) Mode-valued operators such as ROW deliver a pointer to a ddb. This ddb is stored in a block obtained by an allocation. The assignment

```
triple ← ROW(3, INT)
```

stores the value of this pointer, a mode value, into the mode-valued variable triple.

In addition to assignment, there is a second operation, called m-def and denoted by "⇐", used for associating a value with a mode-valued variable. For example, we may write

```
triple ⇐ S(a:INT, b:INT, c:INT)
```

which is roughly equivalent to

```
val(triple) ← val ∘ S(a:INT, b:INT, c:INT)
```

The operator "⇐" takes two arguments of type mode, applies val to the left-hand argument obtaining a ddb, and copies into this ddb the result of applying val to its right-hand argument. The immediate consequence of this operation is that any mode which points to the changed ddb is changed in meaning.

The technique used in EL1 for excluding recursive modes and handling other modes of forward reference can now be discussed. There are four builtin operations for creating modes: ROW, STRUCT, RANY, and PTR. All but PTR require that the modes delivered as arguments be already defined (by a prior assignment or m-def), and that these modes not be of class rany. The first condition is effectively equivalent to the requirement that there be a partial ordering on modes created using ROW, STRUCT, and RANY. For example, our canonical illustration of recursion

```
m1 ← S(a:INT, b:m2) ;
m2 ← S(c:CHAR, d:m1) ;
```

violates this restriction and is illegal. As a second illustration, consider

```
m1 ← S(a:INT, b:m2) ;
m2 ← S(c:CHAR, d:BOOL);
```

This is not admissible as it stands, but the requirement of partial ordering is not violated; hence, it can be re-ordered and rendered legal.

The operator PTR, on the other hand, accepts modes which have not yet been defined, requiring only that they have a non-null value. That is, a mode is an acceptable argument to PTR if either (1) it has been properly defined, or (2) it has been given a dummy initialization. In the latter case, it is said to be weakly defined. Consider, for example,

```
DECL m1, m2 : mode;
m2 ← allocate (ddb, ⟨ ⟩);
m1 ← S(a:INT, b:PTR(m2));
```

The second line allocates a ddb with default values (c.f. §3.14.1) and sets m2 to reference this. The third line uses m2 as an argument to PTR which is legal since m2 is non-null.† PTR produces a mode value which is used as an argument to STRUCT. Continuing with the example, the desired definition is completed with

```
m2 ⇐ S(c:CHAR, d:PTR(m1));
```

The definition of m2 is performed by an m-def. That is, before execution of the line, m2 points to a ddb, the m-def assigns to this ddb the definition appropriate to a struct with two components: an INT named "c" and a PTR(m1) named "d".

To summarize the above discussion, we list the conventions adopted in EL1.

(1) STRUCT, ROW, and RANY require their arguments to be properly defined; recursive modes are thereby excluded from the language.

---

†This presents no problem in implementation: PTR can accept weakly defined arguments because it only requires the locations of the ddbs of its arguments. (c.f. §5.9.5)

(2) Other modes of forward reference can be defined using PTR and m-def. The schema for defining such a mode $\mathfrak{M}$ is as follows.

(a) $\mathfrak{M}$ is given a dummy value by the initialization[†]

$\mathfrak{M} \leftarrow$ allocate (ddb, ⟨ ⟩);

(b) $\mathfrak{M}$ is used as an argument to PTR.

(c) $\mathfrak{M}$ is properly defined by an m-def

$\mathfrak{M} \Leftarrow \mathfrak{F}$

where $\mathfrak{F}$ is the defining form.

3.16 BUILTIN DATA TYPES CONTINUED

In section 3.4 we presented the ten builtin data types (Boolean, integer, character, mode, ptr-any, procedure, none, noneref, symbol, and stack) and discussed the first four. The types ptr-any and procedure were treated in sections 3.9.4 and 3.12. Here, we examine the remaining four: none, noneref, symbol, and stack.

3.16.1 NONE and NONEREF

The mode NONE is the data type of the empty object. For example, a procedure which performs its operation by side effects may be declared to return NONE (i.e., the result-type is NONE, c.f. §3.12). The procedure then returns no value; if there is a value in hand it is thrown away. Occasionally, it is useful to denote the empty object; hence, there is a

[†] Note that we do not want such an initialization to be performed automatically on declaration, for one does not always want to allocate a ddb when creating a mode-valued variable.

constant having it as value, written "NOTHING". For example, the following compound form is exited with no value if p(x) is TRUE

$$[\![x \leftarrow f(y);\ p(x) \Rightarrow \text{NOTHING};\ z \leftarrow q(y)]\!]$$

The mode NONEREF is the data type of the value NIL. This, in turn, is defined as follows. An object of class ptr always has some value; either this is the address of some object of the appropriate type, or it is the value NIL. NIL has the property that if p is any pointer, the assignment p ← NIL is legal. Since p is a pointer, we may ask: to what does p then point? It is useful to adopt the convention that NIL, or a NIL-valued pointer, points to NOTHING. Hence, val(NIL) ≡ NOTHING and mval(NIL) ≡ NONE.

3.16.2 Symbols

The term "symbol" is used with approximately the same meaning as "atom" in Lisp 1.5 [McCar62]. That is, a symbol is a sequence of zero or more characters represented internally by a pointer to a symbol table entry. A symbol constant is written by enclosing the sequence of characters in double quote marks, for example: "symbol", "ANOTHER", "yet*%#159another", and "@b$5←g".

3.16.3 STACKS

The mode STACK designates a class of objects which behave like ordinary LIFO (last-in-first-out) stacks with a few additional properties which make their use "safe". The introduction of stacks into the language is motivated by the existence of a class of algorithms which require dynamic storage allocation but (a) use it in strict LIFO fashion and (b) can perform explicit freeing of unused storage. Such dynamically allocated storage can be obtained using allocate alone, but then there is no way to free these

blocks or to make use of the knowledge that some set of blocks is created and destroyed in LIFO order. A STACK allows an algorithm which uses dynamic storage in a disciplined fashion to gain the efficiency which that discipline permits.

STACKs can best be described by means of an extended example. Consider

```
stack_ptr ⇐ PTR(STACK);
    .
     .
      .
DECL s : stack_ptr;
```

The declaration establishes that s is a variable which can point to a STACK. Next, suppose

```
s ← allocate(STACK, ⟨t⟩);
```

s now points to a STACK t units long. The length of a STACK specifies its capacity for holding objects. It will be convenient to use "$\mathcal{S}$" to denote the STACK to which s points, i.e., $\mathcal{S} \equiv$ val(s). Since nothing has been stored in $\mathcal{S}$, it is currently empty. However, suppose at some later time we execute

```
DECL p1, p2, p3 : PTR_ANY;
p1 ← get_stack_space(s, intρ, ⟨k⟩);
```

The procedure call obtains from the STACK to which s points a block large enough to hold an intρ of length k, initializes the block, and returns a PTR-ANY which references the intρ. The assignment sets p1 to reference the intρ. If next

```
p2 ← get_stack_space(s, string, ⟨n⟩);
```

then $\mathcal{S}$ contains two items: an intρ and a string. The "top" or last element of $\mathcal{S}$ is the one most recently created – here, the string. Subsequent calls

on get-stack-space may create additional elements in the STACK $\mathcal{S}$.

To a first approximation, p1 and p2 behave as if they referenced ordinarily allocated blocks. For example,

```
val(p1) [i]
```

is an INT: the i^{th} element of the intρ in $\mathcal{S}$. Similarly,

```
val(p2) [3] ← 'w
```

is an assignment to the 3^{rd} element of the string in $\mathcal{S}$. Further, the PTR-ANY values may be copied, e.g.,

```
p3 ← p2
```

sets p3 to reference the string so that val(p3) ≡ val(p2). It must be emphasized that p1, p2, and p3 point into $\mathcal{S}$; i.e., to objects created within the STACK. The stack-ptr s, on the other hand, points to the STACK.

Pointers into STACKS are unusual in that the objects to which they point may be destroyed. Continuing the above example, suppose

```
free_last(s)
```

This destroys the last object created within $\mathcal{S}$; i.e., the string. $\mathcal{S}$ now contains a single element, the intρ. The above line not only destroys an object, but also changes the value of all pointers (here, p2 and p3) which reference it to NIL. That is, free-last(s) carries out two functions: (1) destroying the last object in $\mathcal{S}$ so that the space can later be reused, (2) destroying all references to the object so destroyed. After the freeing, val(p2) ≡ val(p3) ≡ NIL; however, the value of p1 is unchanged. A subsequent call "free-last(s)" would make p1 NIL as well.

There is one additional procedure which is useful in operating with stacks: last-in. The form

last_in(p, s)

where p is a PTR-ANY and s is a stack-ptr, is a predicate with value TRUE if and only if p points to the last object created within the STACK to which s points. In the above example, before the string is freed we have last-in(p1, s) ≡ FALSE and last-in(p2, s) ≡ TRUE. After the string is freed, we have last-in(p1, s) ≡ TRUE, last-in(p2, s) ≡ FALSE.

The three procedures get-stack-space, free-last, and last-in are the only operations defined on STACKs. Collectively, they give an operational definition of the mode STACK, indeed, the only definition which need be given.

3.17 MISCELLANEOUS TOPICS

3.17.1 Mode Compatibility

Whenever an attempt is made to associate with an object a value having "incompatible" mode, a type error results. Such associations can be attempted under two principal circumstances: by an assignment, and in binding a formal parameter. Since the notion of "compatibility" is treated somewhat unusually in EL1, a brief discussion is in order.

With a few exceptions involving pointers (c.f. compatible in §5.14), two modes are compatible if and only if they reference the same data definition block, i.e., contain the same address. This makes possible very rapid type checking: essentially, a single address comparison. This also makes it possible to create objects with identical structure but incompatible modes, for example

```
triple ← R(3, INT);
triple_too ← R(3, INT);
  .
    .
      .
DECL x : triple;
DECL y : triple_too;
```

The variables x and y are incompatible and assignments such as "x ← y" are type errors.† In general, this convention produces a desirable result. Distinct names will be used to designate conceptually distinct data types; a given type will be defined only once so that the above problem will not arise. However, it should be noted that two data types may have identical representations yet be treated differently. For example,

```
point ← S(x:INT, y:INT);
pair  ← S(x:INT, y:INT);
```

In most cases, assignment of a pair to a point or passing a pair to a procedure which expects a point would be a conceptual error.

3.17.2 Free Variables

In all the examples thus far, all variables used in procedures have been bound; i.e., either formal parameters or declared variables. A variable which is not bound is said to be free, for example, d in

```
f ← PROC(x:INT) INT; x + d ENDP;
```

When f is executed, its value is the sum of x and d. The value of x is the value of the argument to f; the value of d is defined to be the value of

---

†The form "FOR i ← 1, . . ., 3 DO x[i] ← y[i]" is, however, legal and produces the intended action.

the most recently created bound variable of that name. Here, "recently" refers to chronological order of evaluation. Another way of stating this is that the meaning of free variables in an explicit procedure is that which would be obtained by substituting the text of the procedure in place of the procedure application.

In general, free variables names are identified with objects according to dynamic scoping (as in Lisp). It should be noted that this differs from lexical scoping of free variables which occurs in the λ-calculus, Algol 60, Algol 68, and most other programming languages. It is our contention that the former is the better choice, leading to simpler, more consistent, and more useful languages. However, we will postpone discussion of this point until section 7 where we carry out a general assessment of EL1.

3.17.3 Error Handling

To a certain extent, EL1 is designed to minimize the occurrence of errors, by carrying out the intent of a program even when it is literally incorrect. For example, if p is a pointer to an intρ, "p[i]" is nonsense if construed by a strict evaluator, for p has no components. However, the form will be interpreted in EL1 as "val(p)[i]" to no one's loss. Whenever a moderate dilation of language provides compact and unambiguous notation, it is permitted. For many cases, the best sort of error handling lies precisely in extending the evaluator such that constructs which would otherwise be errors become well-defined. EL1 currently goes some distance in this direction; as experience with the language and its common errors is gained, further latitude will be admitted.

There are, however, some cases in which the evaluator cannot handle a form. All such cases are processed in a uniform fashion. A check is made to see if there is a programmer-defined procedure for dealing with that class of errors. If so, the procedure is called and the value of that procedure is taken to be the value of the form which caused the error. If no such procedure exists, an error message is output and implementation-defined action is taken to effect recovery.

External interrupts will be handled in the same fashion. Although the language presently has no external interrupts, we assume that its application in fields such as graphics will require their addition. When they are added, the interrupt handler will operate like the error handler: it will first check for a programmer-defined procedure for that interrupt class (e.g., "light-pen-interrupt") and failing that will execute a system procedure.

In the case of errors or external interrupts, the programmer-defined procedure is identified by name and may be changed (e.g., by assignment) in the program being evaluated. This avoids need for a special scope rule and control mechanism as, for example, with the ON statement of PL/I [IBM66a].

Section 4. SEMANTIC FOUNDATIONS

In section 3, the language EL1 was informally presented; in section 5, a formal definition will be given. In this section, we discuss a number of topics on which the formal definition depends but which are not formally defined. We thereby supply the foundations on which the formal definition is based. These topics include (1) the representation of programs used by the evaluator, (2) the relation of this to the source text, (3) the storage management system used by the evaluator. We also treat one meta-issue: the linguistic circularity resulting from the method of semantic specification employed in the formal definition.

4.1 ABSTRACT SYNTAX AND ITS RELATION TO CONCRETE SYNTAX

As discussed in section 2.1.3, the notion of abstract syntax was introduced by McCarthy as a method of directly describing those properties of a program which are of interest to an interpreter. McCarthy's scheme used predicates, functions true of specific classes of program objects, and selectors, functions which project out a component of a program object. To these two, Landin [Land64] added a third class of functions, constructors, which create program objects of specified type. Later, the Vienna model of PL/I [Luc68] appropriated the term "abstract syntax" to describe a scheme which allowed only predicates, but with an embellished format suited to the metalanguage employed in the model.

From this diversity of usage, the term "abstract syntax" has come to refer generically to formalisms for describing the underlying structure of a program. It is used specifically in opposition to "concrete syntax" (e.g., context-free grammars, type-2 grammars) which describe the written appearance of a program. As a single concept can be represented in many

notational forms, a single abstract syntax may correspond to many concrete syntaxes, possibly quite dissimilar.† For our purposes, the relevant distinction is that a concrete syntax describes the external representation (i.e., string text) of the source program, while the abstract syntax describes an internal representation more amenable to the process of evaluation.

The external representation of EL1 is a context-free language. This is specified using a formalism which generates the context-free languages but provides a number of augments to the notation of context-free grammars. The formalism will be touched on in this section and treated fully in section 5.1.1. The abstract syntax of EL1 is expressed using the mode definition facilities presented in section 3.9.

In an abstract syntax there are three logical concepts to be expressed:

(1) the notion of compound objects containing a fixed number of components, e.g., a clause (c.f. §3.7) consists of a test which is a form and a consequent which is a form;

(2) the notion of compound objects containing an indefinite number of components, e.g., a compound form consists of zero or more statements;

(3) the notion of alternative formats for a single syntactic type, e.g., a statement is either a form or a clause.

These three concepts are represented by three mode sub-classes: struct, length unresolved row, and united ptr. For example, corresponding to the three illustrations above we have the three abstract syntax definitions

(1) clause ⇐ STRUCT(test:form, consequent:form);

---

†For example, the addition of x and y can be denoted by: x + y, + xy, + (x, y), (PLUS X Y), plus [x; y], etc. Each of these has a different concrete syntax.

(2) compound_form ⇐ ROW(statement);

(3) statement ⇐ PTR(form, clause);

Representation of compound objects containing a fixed number of components is relatively straightforward. The STRUCT definition is not very different from the description in English. Further, this corresponds directly to context-free concrete grammars where the right part of a production consists of a fixed number of elements. For example, the concrete syntax for clause is

clause → form ⇒ form

Here, the concrete and abstract syntaxes differ in only two ways. (1) The components in the definition part are named in the abstract syntax but not in the concrete. (2) Delimiters such as "⇒" which serve as punctuation in the concrete syntax have no counterparts in the abstract syntax.

Turning to the notion of compound objects containing an indefinite number of components, we begin by observing that a special representation is not absolutely necessary. For example, a compound form could be defined as a linked list of statements; such a definition would require only STRUCT and PTR. However, such a representation is at best indirect. A sequence of components logically corresponds to the notion of ROW; hence, the chosen representation.

Regretably, there is no corresponding notion in context-free grammars. Using a context-free grammar, one is forced to use the concrete analogy of a linked list; i.e., a recursive definition such as

compound_form → BEGIN compound_body END

compound_body → empty | statement; compound_body

Such representation is counter-intuitive and becomes quite clumsy when

used with any frequency. Consequently, in our specification of concrete syntax we use a variation on notation† introduced by R. Floyd [Floy63] and write

compound_form → BEGIN {statement ;}⊛ END

which may be read as: a compound form consists of the symbol BEGIN, followed by zero or more statements separated by semicolons; followed by the symbol END. To summarize, an indefinite number of components of the same syntactic type is represented in the abstract syntax by a row, and in the concrete syntax by a special notation using the meta-symbols "{", "}", and "⊛".

Alternative formats for a syntactic type are represented in concrete syntax by alternative right parts of a production, for example,

statement → form | clause

Representation of this notion in the abstract syntax is somewhat indirect. For example, consider the abstract type statement which is either an abstract form or an abstract clause; the potential choice could be represented in Algol 68 (c.f. §2.2.2) by

mode statement = union (form, clause)

This would be a reasonably direct rendering of the English description. However, EL1 does not have the general concept of union. Instead, we interpose a pointer and define a statement to be an object which can point to either a form or a clause.

---

†The meaning of this notation should be clear from context. A precise definition is given in section 5.1.1.

It might appear that the representation we have chosen is wasteful when compared to one based on union. However, this is not the case. Most uses of alternative formats involve recursive definition, either directly or indirectly, in the concrete syntax. In such cases, the union definition will not work, for the syntax recursion implies a recursive mode and a pointer must be used (c.f. §3.15). Even where the union definition could be used it is generally undesirable, for it tends to waste storage – more than that wasted by the pointer. If a statement is represented as the union of form and clause, the storage for a statement must be large enough to hold either of its alternatives; this storage will be required even when the smaller alternative occurs and the remainder will be wasted. The waste results from the binding which a union leaves open even after the relevant choice has been made.

One additional difference between the abstract and concrete syntaxes of EL1 should be noted. The former is expressed in the formalism of the language; the latter is not. Hence, while an abstract program is a legitimate data object of the language, a concrete program is not. The formal definition given in section 5 assumes that all programs have been previously translated from concrete to abstract form.†

The translation is carried out in two phases: (1) parsing the source text into the generation tree specified by the concrete syntax, (2) producing from this generation tree the equivalent object defined by the abstract

---

† We could, of course, define a source program as a string (c.f. §5.5.3), define a generation tree as another data type, and write the parser and translator as procedures in EL1. We have decided against doing so here in order to avoid dilution of this work by issues incidental to its central thesis.

syntax. The first phase is outlined in section 4.3.1; the second is treated in section 5.1.3. Lest a misconception arise, we wish to point out that these two phases are conceptual entities, not chronologically sequential passes. That is, it is not necessary to grow the entire parse tree before producing abstract program; as portions of the tree are completed, they may be individually transformed.

4.2 LINGUISTIC CIRCULARITY

As the meaning of programs written in EL1 is defined by an evaluator also written in EL1, there is a direct circularity in the semantic specification. In this section, we explore the circularity and its consequences.

From a utilitarian standpoint, there is little problem. As pointed out by McCarthy [McCar66] and Minsky [Mins69] such a definition, while circular, is quite useful. To understand the language one needs only to know the workings of a single program, the evaluator, not all implications of all possible programs in the language. By concentrating on a particular program, the complexity is reduced by several orders of magnitude. For example, the language can be communicated (e.g., to an implementor or a standards committee) by means of this particular program. As it is far easier to explicate and deal with a single program than the set of all possible programs, the advantage is non-trivial.

An argument based on utility, however, addresses itself to only part of the problem. There remains a lurking doubt as to whether the circularity is logically admissible. While the doubt cannot be altogether dispelled, we can clarify the issue.

In a strong sense, such circularity or its equivalent is inescapable. To define the meaning of a language £, it is necessary to use a metalanguage £′.

How then to define £′ ? Either (1) £′ is the same as £, (2) £′ is so simple as to require no further specification, or (3) £′ must be defined by a meta^2-language £″. The third choice merely repeats the problem with £ replaced by £′ ; while it may be practically advantageous to do so and thereby descend one or more semantic levels, the logical problem remains unchanged. Hence, circularity can be avoided only if the second alternative is employed.

A definition based on a metalanguage so simple as to require no formal specification presents some difficulties. While such languages do exist, e.g., the order code for a simple Turing machine, they are semantically very remote from those we wish to describe. To bridge the distance, the semantic specification must entail piling many levels of definition, one upon another. Such a tower, while acceptable to an automaton, would be virtually useless for human consumption, so useless that serious doubts concerning its correctness would be justified. It must be appreciated that a semantic specification, like a mathematical proof or a computer program, must be debugged. This entails a referee and imposes the requirement that the specification be intelligible; a non-trivial requirement. Further, the cascading of definitional levels upon a simple base such as a Turing machine has a second difficulty. When applied to a program, the definition may yield the right answers but seriously misrepresent the process whereby these answers are obtained. We postpone to section 7.3 a general discussion of the validity of such misrepresentations. Here, we simply note that such models will give a seriously distorted picture of the language mechanisms and hence will be useless in language design or discussion of design. On the other hand, it is unlikely that a metalanguage much more complex than a Turing machine is truly so simple as to require no formal specification.

In short, there are good reasons to believe that an attempt to define a non-trivial programming language in terms of a self-evident metalanguage will either founder upon the complexity of definition or end up using an unacceptably complex metalanguage, thereby violating its constraints. Hence, a circular definition is unavoidable. The relevant question is therefore not whether to admit a circular definition, but how to make it palatable. Specifically, the issue is where the circularity should lie. One could, for example, define the source language $\mathcal{L}$ in terms of some simpler metalanguage $\mathcal{L}_m$ and $\mathcal{L}_m$ in terms of itself. Alternately, one could employ a symmetric scheme: define $\mathcal{L}$ in terms of a metalanguage $\mathcal{L}_m$ and $\mathcal{L}_m$ in terms of $\mathcal{L}$. Many other schemes and variations present themselves; choice among these depends in large measure on the language being defined.

In the case of EL1, there are compelling reasons for the use of direct circularity. As a base for an extensible language, EL1 is to be as simple as possible, consistent with the goal of spanning a certain semantic space. If EL1 could be satisfactorily defined in terms of some simpler language $\mathcal{L}_m$ then $\mathcal{L}_m$, not EL1, would be the appropriate base language. With such considerations in mind, we designed EL1 to be close to the minimal fixed point of its space. Notions which could be specified in terms of other notions were, with few exceptions, excluded. These exceptions are primarily embellishments added to EL1 so that it serves as a fluent metalanguage for its self-description.

Several useful consequences arise from this self-description. When learning the language, one learns language and metalanguage simultaneously, avoiding the significant difficulty of learning two new languages at once. Also, when metaphrase extensions (c.f. §1) are made, they are written in

the language itself with obvious attendant benefits.† Finally, identity of language and metalanguage has implications in the field of proving properties of programs. Suppose we have strong proof techniques and are able to prove properties of algorithms expressed in the language. These proof techniques can be used in particular to prove properties of the evaluator and hence of the language as a whole. Such proofs may be an additional handle on language semantics, providing a static counterpart to the procedural and therefore dynamic evaluator.

4.3 THE UNDERLYING SYSTEM

4.3.1 Parsing

The first phase in translating an EL1 source program into internal representation entails parsing it into its generation tree as specified by the concrete syntax. The problem of parsing has been one of the most intensively studied areas of computer science and reasonably satisfactory techniques are known. Consequently, we shall only outline the parse method and refer the interested reader to the relevant literature.

Parsing may be conceptually divided into two stages: lexical analysis and syntactic analysis. (See [Chea67] for a complete discussion of this division and the place of these two stages in a compiling system.) Lexical analysis consists of accepting a string of characters, breaking this string into tokens (e.g., identifiers, numbers, and delimiters), and outputting a

---

†The reader who questions the significance of such benefits is invited to contrast the operator definitions of MAD (c.f. pp. 104-107 of [Ard64]) as given in pseudo assembly code with an equivalent rendering in MAD. While the definitions are quite transparent when written in MAD, they present a formidable intellectual challenge in pseudo assembly code.

sequence of token descriptors. Syntactic analysis consists of identifying syntactic units in the token descriptor sequence and constructing some suitable representation of the parse tree. These two stages can in principle be rendered as two sequential passes, but it is far more efficient to use two procedures connected as coroutines. (For a discussion of coroutines, see section 1.4.2 of [Knu68].) In the latter case, the syntactic analyzer calls upon the lexical analyzer each time it requires a token.

Construction of a lexical analyzer from a specification of concrete syntax is discussed in [John68]. Only one point requires discussion here, that being the form of token descriptors. In internal representation, an identifier is represented by a pointer to a symbol table entry (c.f. §5.5.3). Since the symbol table is available at all times, it can be employed in parsing; the descriptor of an identifier is a pointer to its symbol table entry. The descriptors of tokens belonging to other classes are pointers to tables used only in parsing. For example, the descriptor for a constant is a pointer to a literal table which contains the type and value of that constant.

We intend that syntactic analysis be carried out using Earley's algorithm [Earl68]. EL1 as it currently stands could be parsed by a number of faster special-case methods (c.f. [Chea67] or [Feld68] for a discussion of such methods) or by a hand-tailored algorithm. However, it is our intention that a facility for syntactic extension be added to EL1 in constructing a complete language core (c.f. §9.2). This facility will be quite general; i.e., any extension to the concrete syntax which can be expressed as a context-free language will be acceptable. Hence, we will eventually require a completely general context-free parse algorithm; it seems wise to provide for the general case at the outset.

Of those algorithms able to parse any context-free language, Earley's appears to be the best (c.f. chapter 16 of [Earl68]). Hence, we have elected to use it. As the concrete syntax of EL1 is quite simple, the algorithm will run in time proportional to the length of the program, with a reasonable constant of proportionality.

4.3.2 Storage Management†

In section 3, the issue of storage classes appeared several times but was glossed over in the interest of initial simplicity. In particular, no distinction was made between different types of storage and storage behaviors. Here we take up the issue of storage classes and discuss their implementation.

An object in EL1 belongs to one of three storage classes:

(1) stack-objects, for example the x in

```
DECL x : int;
y ← x;
```

(2) heap-objects, for example the value of val(p) in

```
p ← allocate(intρ, ⟨ ⟩);
v ← val(p);
```

(3) pure-values, for example the value of

```
y + 1
```

The first two of these are referred to generically as proper-objects.

---

†It should be pointed out that this discussion is largely confined to storage management for objects which appear explicitly in the language. System objects such as I/O buffers and the interpreter code are dealt with only in passing.

The three classes are defined and handled as follows:

(1) DECLared variables and most[†] formal parameters are stack-objects. Such objects exist only during activation of the procedure in which they are created; when the procedure exits, the objects are destroyed. Hence, their storage can be managed as a LIFO (last-in-first-out) stack.

(2) Objects created by a call on the procedure allocate are heap-objects. They exist so long as any pointer references them. Dynamic allocation from a storage pool provides blocks to hold these objects; when necessary, garbage collection returns to the pool those blocks no longer referenced.

(3) A pure-value is an ephemeral object, momentarily arising as the result of some calculation such as "y * x", or "determinant(a)". Since the evaluation of an EL1 program takes place serially, there can be at most one pure-value at any given time. Hence, pure values reside in a single block of storage which is used as a degenerate LIFO stack holding at most one value. Whenever a pure-value is created, any previous pure-value is destroyed.

In terms of spanning a semantic space, heap-objects form the most general storage class and subsume the others. We could abolish the other two classes and implement their objects as heap-objects with no loss of

---

[†]Specifically, a formal parameter is always stack-managed under either of the following conditions: (1) it is declared to be bound BYVALUE or UNEVALED, (2) the argument is a pure-value. The remaining case — when the binding is declared BYREF and the argument is a proper-object — is sometimes stack-managed. In this case, the binding merely links the formal name to the argument; hence the storage class of the formal parameter is that of the argument object. This may be either stack or heap, depending on how the argument was created.

language features. Instead of explicit destruction of objects at procedure exit, the system would merely wait for a garbage collection. The two stack classes are introduced only to gain efficiency. However, these gains are considerable. Compared to use of a stack, garbage collection is slow, particularly when memory becomes full (for example, c.f. section 2.3.5 of [Knu67]). For those variables which are created and destroyed in strict LIFO order, the use of a garbage collector invokes a needless waste one can ill afford.

This holds a fortiori in a two-level storage system; i.e., where the address space is not held entirely in core but rather kept on a bulk device and brought into core in segments by software or hardware paging. Time required to access data on a segment not in core is typically four orders of magnitude greater than the time for an in-core access [Coh67], [Bobr67a]. Hence, garbage collection is slower and the storage fragmentation induced by a garbage collection system, but avoided by a stack, is more costly. Use of a stack insures that all objects having the same scope reside in a contiguous portion of the address space. Hence, they will be brought into core together. Since they will in general be used together, the use of a stack causes an effective compactification.†

Implementation of a LIFO stack, either for the stack-objects or the pure-values, is relatively straightforward and will not be discussed here. (The reader may consult [Dijk60] and [Ros66].) We shall, however, examine the more difficult issue of heap implementation.

---

† We note in passing that the benefits of additional compactification may in many cases be a good reason to pass arguments BYVALUE (c.f. §3.12) when their lengths are smaller than the segment size. If this cuts down the working set sufficiently that it fits into core where otherwise it would not, the cost of copying is far outweighed by the savings produced by not making out-of-core segment references.

The basic technique is simple. Some part of the address space is used for the heap; storage in the region is said to be initially free. All requests for heap storage arise from calls on allocate, e.g., "allocate(m, d)" where m is a mode and d is an intρ used as a dope-vector. From m and d the number, n, of storage units required is computed.† A block of n units is removed from the free region and a pointer to it is returned as the value of allocate. The block is then said to be reserved. After some time, the heap will have become completely reserved by this process. It will be found that the next request for storage cannot be satisfied. This causes a garbage collection which reclaims all blocks which were earlier reserved but are no longer referenced, i.e., effectively no longer in use.

The first point to be noted is that there are several ways in which the address space allotted to the heap can be obtained. (Note, also, that it need not be a contiguous block.) The region can be either fixed as in CORAL [Rob65] and AED [Ross67], or it can be allowed to grow by making calls upon a higher level storage allocator. In the latter case, the higher level allocator may be either the operating system or a global allocator for the sub-system. A fixed region scheme is simplest to implement but has little else to recommend it. Different programs will make widely differing demands upon the heap size, demands which will in general be known only during program execution. Hence, there is strong motivation to allow heap growth, particularly since randomly scattered blocks are usable in forming the heap. Calls upon the operating system for additional storage make sense

---

†Note that this quantity is implementation dependent, for it is a function of the bit pattern representation of the primitive data types. It is not even the case that $n_1 > n_2$ in one implementation implies $n_1 > n_2$ in all.

only when the address space allotted to a sub-system is less than its potential address space. This may occur in either an in-core address space as on an IBM 360 [IBM66b], or a two-level memory system as in Lisp 1.85 [Bobr68]. When the condition holds and calls upon the operating system are possible, this method allows the most flexible distribution of resources to competing tasks. When this is not possible, global storage management by the sub-system allows some trading between the heap and other pools, resulting in somewhat better performance than the fixed region scheme.

One critical issue of heap management is how a free block of the desired size, say n storage units, is to be found. After a number of garbage collections, the free blocks of the heap will be scattered throughout its extent. It is necessary that the heap be so organized that a block of size n can be readily found whenever such a block exists. There are several techniques for this organization.

Knuth [Knu68] discusses use of a single list of all free blocks, sorted according to starting address. To obtain a block of size n, the list is searched until a block of size $m \geq n$ is found. If $m > n$, the block is split and the excess is returned to the free storage list; if $m = n$, the block is used directly. To spread out the appearance of fragments split off from larger blocks by this process, search to satisfy a request is started not at the head of the free list but at the point where the previous search stopped.

This scheme is designed to allow efficient operation in a system where blocks can be explicitly freed; a single, sorted list is used so that a freed block can be merged with the blocks above and below it, in the event that these blocks are free. Where, as in EL1, explicit freeing is not allowed this scheme can be improved by keeping separate lists for various block sizes, say one list for all blocks falling between each power of two; this,

of course, cuts down the search. A fragment split off from a block larger than the request size is placed on the list appropriate to its size.

It is possible to go further and completely avoid search for a sufficiently large free block by rounding up all requests such that they become a power of two. Satisfying a request is then carried out by taking the first block from the appropriate list or, should this list be empty, breaking a block from the next higher list in half. This scheme may be attractive in a two-level storage system where a search may entail several expensive references to secondary store as the pointer chain is followed. However, unless the address space is very large, the waste produced by rounding up (typically, 25%) will make this too expensive. Also, it has the further disadvantage that with this scheme it is difficult for the garbage collector to reclaim part of block. Hence, for our purposes, the multilist scheme without rounding up is preferable.

The AED free storage package [Ross67] uses a rather different technique. The heap is divided into zones, each having its own allocation strategy and management technique. Zones may be divided into sub-zones, called sons, in standard hierarchial fashion with usual genealogical terminology for the relations. The use of zones is intended to gain efficiency from exploiting a common phenomenon: a program typically uses only a small number of modes and has few distinct block sizes which are used with great frequency, some only during specific time spans. If each block size is handled by a specific zone, there is less expense due to breaking and recombining blocks and less problem of storage fragmentation. The AED system provides explicit and detailed control over the creation of zones, the strategies they use for storage allocation and compactification, and the allocation of blocks from specific zones. While most of this control is too

fine to expose in EL1, much of the AED strategy could be used hidden behind the simple call on allocate. It might be possible for the heap management system to keep statistics on the program, determine what block sizes are in use, and set up special zones for the most common sizes.† Whether the savings induced by zoning pay for the mechanics of creating it can only be decided by experience and will depend very heavily on the program being run. We assume that implementations will initially adopt the single-zone-multilist scheme and add provision for special zones as need for them arises.

The next point to be considered is gargage collection. In outline, this works as follows. Starting from base positions (e.g., result-slot and value-stack, c.f. §5.3.5) all pointer chains into and within the heap are traced, and all objects so encountered are marked. Since these objects are not homogenous, and the data type of an object is not stored with the object, it is necessary to keep an address and a mode for each pointer. (In EL1 terms, the tracing is carried out using PTR-ANYs.) This is straightforward, for one can immediately determine the mode of objects in the base positions; further, given the mode of an object $\mathcal{O}$, one knows the modes of its components and therefore the modes of the objects to which the components (or $\mathcal{O}$ in the case of a single pointer) may point. The only non-trivial point

---

†One particularly nice technique for implementing a (non-hierarchial) zone system is a quantum map. The address space is divided into segments whose size is some power of 2. A table, called the quantum map, is kept in which the i^{th} entry describes the i^{th} segment of the address space. In particular, the descriptor may include a zone type. The point of this arrangement is that given a pointer the descriptor and hence the zone type may be accessed in but a few machine operations (e.g., load the pointer into an index register, right shift, and fetch indexed). Note also that the quantum map may be used in global storage management, some segments being allotted to the heap, some allotted for buffers, and others unallotted.

is that row modes may be length unresolved, in which case it is necessary to access the length field(s) which is stored with the object.

In marking objects as being in use, it is necessary to use a bit table for those objects (e.g., integers) which occupy a full word. Garbage collection is made easier if the bit table is used for all objects. Each word of the heap is represented by a bit; marking an n-word object as being in use entails setting n bits to 1.

Having marked an object $\mathcal{O}$, the garbage collector traces all pointers in $\mathcal{O}$. While tracing the descendants of $\mathcal{O}$ stemming from one pointer, it is necessary to remember $\mathcal{O}$ so that the garbage collector can later return to it and trace from the other pointers. There are two methods of doing this, depending on whether $\mathcal{O}$'s are remembered by (1) using a special stack, or (2) reversing pointers using the method of Schorr and Waite (c.f. section 2.3.5 of [Knu68]). By omitting the need for reserving a special stack, the latter allows a somewhat larger heap. However, as the number of objects to be so remembered rarely gets large in practice, the gain may be negligible. The objection to the second method is that it runs about four times slower than the stack technique [Sch67] because it visits each node several times as often.

In view of its superior speed, we have chosen to use the stack method. Before tracing a pointer $\mathcal{P}$ belonging to an object $\mathcal{O}$, the garbage collector stacks a triple consisting of the location of $\mathcal{O}$, its mode, and which pointer in $\mathcal{O}$ will be traced next, if any. When all the descendants of $\mathcal{P}$ are processed, the stack is popped and tracing continues with the next pointer in $\mathcal{O}$.

Having thus marked all structures in the heap which are referenced, the garbage collector enters its second phase. It makes a linear sweep of the bit table: a sequence of n $\mathcal{O}$'s corresponds to a sequence of n

unreferenced words. Each such block is returned to the free storage pool by linking it into the list of free blocks appropriate to its size. Its actual length (i.e., n) is recorded in the first word of the block, except in the case of single and double word blocks.

It will be noted that the above garbage collector does not perform compactification, i.e., moving objects while altering pointers to preserve topology so that the free space becomes one contiguous block. By not compactifying, we run the risk of failing to satisfy an allocate request because the free storage while great enough in total is fragmented into blocks, each too small to satisfy the request. Simulations [Rand69] and [Rob65] indicate that under conditions involving blocks of random size, the effective loss in storage due to fragmentation is on the order of 10 to 15%. To the extent that in practice blocks tend to fall into a small set of fixed sizes, this figure will be correspondingly lower. The use of a zone mechanism would cut this still further. While 10 to 15% is hardly negligible, such a loss may be acceptable, particularly in early implementations. It may prove desirable to later add a compactifying garbage collector using the technique employed in Lisp 2 [SDC67]. Alternatively, if the address space is extremely large, it may be profitable to divide the heap into two semi-spaces, use only one at a time, and compactify by copying from the one in current use to a contiguous region of the other, as in [Fen69].

Section 5. THE FORMAL DEFINITION OF EL1

This section gives the formal definition of the programming language EL1. The written representation used throughout is that of the reference language. Implementations may choose to use a different hardware representation by modifying the concrete syntax (c.f. §5.1.1).

Section 5.1 explains the notation used in defining the language. Some preliminary issues concerned with written representation are treated in section 5.2. Sections 5.3 to 5.13 present the various constructions of the language; auxiliary routines used in these sections are listed in section 5.14. Finally, sections 5.15 and 5.16 list primitive and builtin procedures of the language.

5.1 FORMALISM FOR THE DEFINITION OF EL1

5.1.1 Formalism for Concrete Syntax

The concrete syntax specifies the external or written representation of the language. It is defined using a formalism related to context-free grammars. The reader may consult any of the standard references (e.g., [Gins 66] or [Feld 68]) for a discussion of the latter; an elementary knowledge of this material will be assumed. We shall use the basic definitions and notation found in these sources with one exception: we denote the full vocabulary, terminal vocabulary, and nonterminal vocabulary by $\mathscr{V}$, $\mathscr{V}_T$, and $\mathscr{V}_N$.

The formalism we employ extends the usual notation for context-free grammars to permit more compact and readable definitions. However, the

extension is in notation only; the formalism generates precisely the context-free languages. The differences from standard context-free grammars are as follows:

(1) Nonterminal symbols may be denoted by strings of characters. Hence, all symbols in $\mathscr{V}$ are delimited by blanks on either side.

(2) A production of the form

$$B \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_n$$

where $B \in \mathscr{V}_N$ and $\alpha_i \in \mathscr{V}^*$ for $i = 1, \ldots, n$ may be written as an abbreviation for the set of productions

$$B \rightarrow \alpha_1,\ B \rightarrow \alpha_2,\ \ldots,\ B \rightarrow \alpha_n .$$

The nonterminal, B, defined by such a production is said to be the left part; the α_i's are said to be alternative right parts.

(3) A production of the form

$$B \rightarrow \alpha_1 \{ \alpha_2 \} \alpha_3$$

is an abbreviation for the productions

$$B \rightarrow \alpha_1\ C\ \alpha_3$$
$$C \rightarrow \alpha_2 | \varepsilon$$

where C is a new nonterminal used in no other productions and ε is the empty string.

(4) A production of the form

$$B \rightarrow \alpha_1 \{ \alpha_2 \}^*\ \alpha_3$$

is an abbreviation for the productions

$$B \rightarrow \alpha_1\ C\ \alpha_3$$
$$C \rightarrow \varepsilon | \alpha_2\ C$$

where C is a new nonterminal.

(5) A production of the form

$$B \rightarrow \alpha_1 \{ \alpha_2 \}^{+} \alpha_3$$

is an abbreviation for the productions

$$B \rightarrow \alpha_1 \; C \; \alpha_3$$

$$C \rightarrow \alpha_2 \mid \alpha_2 \; C .$$

(6) A production of the form

$$B \rightarrow \alpha_1 \{ \alpha_2 \, a \}^{\circledast} \alpha_3$$

where $\alpha_1, \alpha_2, \alpha_3 \in \mathscr{V}^*$, $a \in \mathscr{V}_T$, and $B \in \mathscr{V}_N$, is an abbreviation for

$$B \rightarrow \alpha_1 \; C \; \alpha_3$$

$$C \rightarrow \varepsilon \mid \alpha_2 \mid \alpha_2 \; a \; C .$$

(7) A production of the form

$$B \rightarrow \alpha_1 \{ \alpha_2 \, a \}^{\oplus} \alpha_3$$

where $\alpha_1, \alpha_2, \alpha_3 \in \mathscr{V}^*$, $a \in \mathscr{V}_T$, and $B \in \mathscr{V}_N$, is an abbreviation for

$$B \rightarrow \alpha_1 \; C \; \alpha_3$$

$$C \rightarrow \alpha_2 \{ a \} \mid \alpha_2 \; a \; C .$$

To summarize, the formalism uses eight special marks: $\rightarrow$ $\mid$ $\{$ $\}$ * $^+$ $^{\circledast}$ $^{\oplus}$. The right-pointing arrow is used, as in standard context-free grammars, with the meaning: rewrites to. The vertical bar indicates alternative right parts of a production. The braces are used to group portions of a right part, possibly in conjunction with the other four marks. The raised star means: none or more. The raised plus means: one or more. Enclosing either of these in a circle indicates that the final symbol of a substring is optional.

5.1.2 Examples of the Formalism for Concrete Syntax

(a) The grammar given by

nest → a nest d {d} | c

generates the language

$\{ a^n \, c \, d^m \mid n \leq m \leq 2n \}$.

(b) The grammar given by

item → {q r}$^+$

generates the language whose strings are

q r q r q r q r q r q r . . .

(c) The grammar given by

block → ({ segment , }$^{\circledast}$)

segment → b

generates the language whose strings are

( ) (b) (b,) (b,b) (b,b,) (b,b,b) . . .

5.1.3 Formalism for Abstract Syntax

The abstract syntax specifies the internal representation of programs. It is defined using the data-type definition mechanism of EL1 (c.f. §3.9). In general, to each definition $\mathscr{D}$ of a nonterminal $\mathscr{T}$ in the concrete syntax there corresponds a definition $\mathscr{D}^{\#}$ of some data-type $\mathscr{T}^{\#}$ in the abstract syntax.

On input to the evaluator, the external representation of a program is transformed into its internal representation as follows. Let $\mathscr{F}$ be an instance of a concrete nonterminal $\mathscr{T}$. $\mathscr{F}$ is first parsed into its generation tree; then this tree is mapped into a data object $\mathscr{F}^{\#}$ of type $\mathscr{T}^{\#}$. The object $\mathscr{F}^{\#}$ is the internal representation of the external object $\mathscr{F}$.

In general, a generation tree headed by $\mathcal{T}$ can be mapped into an object $\mathcal{F}^{\#}$ of type $\mathcal{T}^{\#}$ by recursively mapping the branches of the tree into the components of $\mathcal{F}^{\#}$. More specifically, this mapping is determined by the correspondence between the abstract definition $\mathcal{D}^{\#}$ and the concrete definition $\mathcal{D}$. The following schemata are usually employed:

(1) A production of the form

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \qquad n \geq 2$$

usually corresponds to the definition

$$A^{\#} \Leftarrow PTR(\alpha_1^{\#}, \alpha_2^{\#}, \ldots, \alpha_n^{\#}).$$

The i^{th} alternative in the concrete syntax is represented by the abstract type $PTR(\alpha_i^{\#})$.

(2) A production of the form

$$A \rightarrow X_1\ X_2 \ldots X_n \qquad (X_i \in \mathcal{V})$$

usually corresponds to

$$A^{\#} \Leftarrow S(N_1 : X_1^{\#},\ N_2 : X_2^{\#},\ \ldots,\ N_n : X_n^{\#})$$

where the N_i's are names for the components.

(3) A portion of a concrete right part having the format

$$\{X\}^* \qquad \text{where } X \in \mathcal{V}$$

usually corresponds to

$$R(X^{\#}).$$

Usually, the correspondence between concrete and abstract syntax is obvious. Where it is not, the correspondence is explained in a note.

5.1.4 Formalism for the Evaluator

The evaluator specifies the meaning of the language. The evaluator $\mathcal{E}$ for a concrete syntactic type $\mathcal{T}$ specifies how instances of that type are to be evaluated. Specifically, $\mathcal{E}$ takes as input an object of type $\mathcal{T}^{\#}$ and

delivers as output a pointer to the result of evaluating that object, i.e., a pointer to its value.

The meaning of a program in the language is specified by the evaluator for the syntactic root program. This calls evaluators for simpler syntactic types which in turn call each other and the evaluators for primitives. The evaluator for each of the primary syntactic forms is listed with the concrete and abstract syntax for that form in sections 5.3 to 5.13. The evaluators for the linguistic primitives are discussed in section 5.15.

The evaluators are themselves procedures written in the language EL1. Wherever practicable, they avoid advanced features of the language and perform their operation as clearly and simply as possible. Consequently, they are in general not locally optimized, particularly with regard to common subexpressions and the occurrence of loop invariant computation within a loop. Where it was thought that a point might be unclear, a comment (c.f. §5.2.3) was added. Comments always appear before the procedure text to which they apply.

5.1.5 Initial Values Used by the Evaluators

The mode constants for the primitive data types in EL1 are denoted by strings of upper-case characters, for example: INT, BOOL, CHAR (c.f. §5.9.1). In the text of the evaluators, it has proved desirable to minimize use of upper-case characters so as to avoid "shouting". Hence, identifiers (written in lower case) having equivalent spelling are defined, initialized to the primitive mode values, and used in place of the mode constants. That is, the evaluators use the following initialized identifiers:

```
DECL int, bool, char, none, ptr_any: mode;
int ← INT;
bool ← BOOL;
char ← CHAR;
none ← NONE;
ptr_any ← PTR_ANY;
```

Also, the evaluator (§5.3.5) assumes that three integer-valued variables, name-pdl-length, value-stack-size, and result-slot-size, exist having values which specify the size of three system stacks.

5.2 WRITTEN REPRESENTATION OF PROGRAMS – PRELIMINARIES

5.2.1 Character Set

digit → 0|1|2|3|4|5|6|7|8|9

upper_case_char → A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|
S|T|U|V|W|X|Y|Z

lower_case_char → a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|
u|v|w|x|y|z|ρ|_

break_char → (|)|'|.|[|]|;|⇒|⟦|⟧|:|⟨|⟩|⇐|,|@

separator_char → ϭ

special_char → =|≠|≥|>|<|≤|+|-|*|/|←|°|∨|∧|!|#|$|%|&|?|↑

string_char → digit|upper_case_char|lower_case_char|break_char|
separator_char|special_char

character → string_char|"

5.2.2 Blanks

A blank (denoted above by "␣") is a separator character. That is, its appearance separates two syntactic units (e.g., identifiers or numbers – c.f. §5.4.1, 5.5.1) but it is not itself a syntactic unit. Other than serving as a separator, blanks have no significance and may be used freely to enhance readability.

5.2.3 Comments

A comment may be inserted into a program by enclosing text between the brackets "COMMENT" and ";". The delimiter "NT" may be used as a left bracket in place of "COMMENT". For example:

COMMENT A comment begins with either the delimiter "COMMENT" or the delimiter "NT" and ends with a semicolon;

NT A comment cannot contain an embedded semicolon;

The delimiter "ELSE" also serves as a comment; it may be inserted wherever useful to make clear the meaning of a program.

A comment acts as a separator character, like blank. Consistent with this convention, comments may appear anywhere in a program.

5.3 PROGRAMS, FORMS, AND THE ENVIRONMENT

5.3.1 Concrete Syntax

```
program → form

form → form2 binary_operator form | form2

form2 → constant | identifier | compound_form | iteration | m_form |
        selection | aggregate | procedure_application | ( form )
```

5.3.2 Examples

Refer to §5.i.2 for i = 4, . . . , 13.

5.3.3 Abstract Syntax

```
form ⇐ PTR(binary_operation, constant, symbol,
           compound_form, iteration, m_form,
           selection, aggregate, procedure_application);

program ← form;
```

5.3.4 Auxiliary Mode Definitions

```
name_pdl_element ⇐ S(name: symbol, old_index: int, datum: ptr_any);
pdl ⇐ R(name_pdl_element);
stack_ptr ⇐ PTR(STACK);
```

5.3.5 Evaluator

```
ev_program ←
PROC (f: form) ptr_any;
DECL name_pdl: pdl SIZE ⟨name_pdl_length⟩;
DECL pdl_index: int;
DECL value_stack, result_slot, aux_result_slot: stack_ptr;
value_stack ← allocate(STACK, ⟨value_stack_size⟩);
result_slot ← allocate(STACK, ⟨result_slot_size⟩);
aux_result_slot ← allocate(STACK, ⟨result_slot_size⟩);
pdl_index ← 0;
install_initial_environment(name_pdl, pdl_index, value_stack);
eval (f);
ENDP;

eval ←
PROC (f: form) ptr_any;
DECL m: mode;
m ← mval (f);
NT There is a separate evaluator for each of the nine types of forms;
m = constant ⇒ ev_constant ∘ val(f);
m = symbol ⇒ ev_symbol ∘ val(f);
m = binary_operation ⇒ ev_binary_op ∘ val(f);
```

m = compound_form ⇒ ev_statementρ ∘ val(f);

m = iteration ⇒ ev_iteration ° val(f);

m = m_form ⇒ ev_mform ∘ val(f);

m = selection ⇒ ev_selection ° val(f);

m = aggregate ⇒ ev_aggregate ° val(f);

m = procedure_application ⇒ apply ° val(f) ENDP;

NT Primitive procedures such as allocate, mval, and val are discussed in § 5.15;

5.3.6 Discussion

A program is a form which is not part of another form. The evaluation of a program entails:

(1) creating and initializing the structures required by the evaluation process,

(2) initializing the environment to contain the definition of builtin procedures,

(3) evaluating the form.

There are three principal structures employed by the evaluators: the name-pdl, the value-stack, and the result-slot. For each variable created in the process of evaluation an entry is made on the name-pdl containing the variable's name, a pointer to its value, and some other information. The value of a variable is stored on the value-stack which is a block of storage managed in LIFO order. The result-slot is used as a register — to temporarily hold a created value. A value so held is said to be a pure value. Storage other than these three structures is referred to collectively as the heap.

Evaluation of a form consists of determining the type of form and, based on this type, calling one of the lower level evaluators.

5.4 CONSTANTS

5.4.1 Concrete Syntax

constant → bool_constant | int_constant | char_constant | noneref_constant |
none_constant | symbol_constant | mode_constant | proc_constant

bool_constant → TRUE | FALSE

int_constant → {-} {digit}$^{+}$

char_constant → ' character

noneref_constant → NIL

none_constant → NOTHING

symbol_constant → "{string_char}$^{+}$"

Cross Reference

character	–	§5.2.1
mode_constant	–	§5.9.1
proc_constant	–	§5.12.1
string_char	–	§5.2.1

5.4.2 Examples

int_constant: 7 10 940 1604 -360

char_constant: 'a '→ ': ''

symbol_constant: "temp" "derivative" "Fourier" "a = b * c" "f"

5.4.3 Abstract Syntax

constant ⇐ PTR(bool, int, char, noneref, symbol, mode, none, proc_var);

5.4.4 Evaluator

```
ev_constant ←
PROC (c : constant) ptr_any ;
NT A constant is treated as a 0-ary procedure. The value of a constant is
   obtained by copying the constant into the return-slot and returning a
   pointer to this copy;
return_result (c , mval(c) )   ENDP ;
NT Auxiliary routines such as return-result are discussed in § 5.14;
```

5.4.5 Discussion

A constant is a form having constant value. It is treated as a 0-ary procedure which always delivers the same result. Predefined constants exist for most of the builtin data types.

Each constant has two representations: an external representation given by the concrete syntax and an internal representation given by the abstract syntax. The external representations should be self-explanatory. The internal representations, it should be noted, involve one extra level of pointers. For example, an int-constant, e.g., 3, is represented internally by a pointer to an INT which contains the bit-pattern for 3. Similar internal representation is used for each type of constant except NONE. The none-constant NOTHING is represented internally by a pointer whose value is NIL.

Note that the noneref-constant NIL is represented internally by a pointer to a noneref which contains the bit-pattern for NIL. Hence, the two forms "val(NIL)" and "NOTHING" have identical values (i.e., evaluate to identical results).

5.5 IDENTIFIERS

5.5.1 Concrete Syntax

lower_case_or_digit → lower_case_char | digit

identifier → lower_case_char {lower_case_or_digit}* | {special_char}+

5.5.2 Examples

x

pressure

a23

*

+ ←

> >

5.5.3 Abstract Syntax

```
symbol ⇐ PTR(symbol_table_element);
string ⇐ R(char);
symbol_table_element ⇐ S(print_name: PTR(string),
                         datum: ptr_any,
                         pdl_position: int);
```

Relation of Abstract to Concrete Syntax

On input, an identifier is converted to its internal representation — a symbol. Symbols are handled in the same fashion as atoms in Lisp 1.5 [McCar62]. Which is to say that all identifiers of the same spelling are represented internally by the same symbol value, i.e., by pointers to a unique symbol-table-element.

5.5.4 Evaluator

```
ev_symbol ←
PROC (name : symbol) ptr_any ;
DECL  p : ptr_any ;
p ← val(name) . datum ;
p  =  NIL ⇒ error ("unbound_identifier") ;
ELSE  p ;
ENDP ;
```

5.5.5 Discussion

Identifiers are the external designations of objects in the language. These objects include not only integers, Booleans, and the like, but also procedures and binary operators.

The value of an identifier is pointed to by the "datum" field of its symbol-table-entry. This is updated and restored as variables of that name are created and destroyed on procedure entry and exit (c.f. §5.13.5). If no variable of a given name has been created, the "datum" field has the value NIL. It should be recalled that the value NIL is the internal representation of the constant NOTHING. Hence, the value of an undefined identifier is NOTHING.

5.6 BINARY OPERATIONS

5.6.1 Concrete Syntax

```
form → form2  binary_operator  form
binary_operator → identifier
```

The following identifiers have been given builtin values as binary-operators:

= ≠ ≥ > < ≤ + - * / ← ∘ ∨ ∧

Refer to sections 5.15 and 5.16.

5.6.2 Examples

```
a + b

x ≥ y * 5

b ← p ∨ q ∧ r

g ∘ x ← y
```

5.6.3 Abstract Syntax

```
binary_operation ⇐ S(lhs: form, op: symbol, rhs: form);
```

5.6.4 Evaluator

```
ev_binary_op ←
PROC (b: binary_operation) ptr_any;
b.op = "←" ⇒ assign (b.lhs, b.rhs);
b.op = "∘" ⇒ apply2(checkproc ∘ eval(b.lhs), ⟨formρ : b.rhs⟩);
ELSE apply2(checkproc ∘ ev_symbol(b.op), ⟨formρ : b.lhs, b.rhs⟩);
ENDP;

checkproc ←
PROC (p: ptr_any BYVALUE) procedure_block;
[[m val(p).class = "ptr" ⇒ p ← val(p) ]];
m val(p) = explicit_procedure ⇒ val(p);
m val(p) = code_procedure ⇒ val(p);
ELSE error ("undefined_procedure");
ENDP;
```

```
assign ←
PROC (lhs: form, rhs: form) ptr_any;
DECL left, right: ptr_any;
DECL pv_flag: bool;
left ← eval (lhs);
[[ pure_value (left) ⇒ BEGIN left ← NIL; pv_flag ← TRUE END ]];
right ← eval (rhs);
pv_flag = FALSE ⇒ assign2 (left, right);
right;
ENDP;
```

Cross Reference

apply2	–	§5.13.4
assign2	–	§5.14

5.6.5 Discussion

In concrete representation, all binary operations are given equal precedence and all associate to the right. The evaluation of a binary operation distinguishes between three cases: (1) assignment, (2) application (denoted "∘") of a procedure to a single argument, and (3) application of a normal infix operator to its two arguments.

Assignment has one special case: if the left-hand value is a pure value, then the assignment is not performed. However, even in this case, the right-hand form is evaluated, thereby assuring that desired side effects will occur.

Procedure application written in the format "⟨proc⟩ ∘ ⟨arg⟩" is syntactic sugar for procedure application written in the format "⟨proc⟩(⟨ arg⟩)". The evaluator reduces the former to the latter.

A normal infix operator is defined by a procedure – either builtin or programmer-defined. The evaluator first obtains the defining procedure and then calls on the normal routine for performing procedure application (apply2). The builtin binary operators are discussed in sections 5.15 and 5.16.

5.7 COMPOUND FORMS

5.7.1 Concrete Syntax

statement → form | form ⇒ form

compound_form → BEGIN { statement ; }$^{\circledast}$ END | ⟦ { statement ; }$^{\circledast}$ ⟧

5.7.2 Examples

```
BEGIN
x ← 0;
FOR y ← 1, ..., n  DO  x ← x+f(y);
END
```

```
⟦ x > y ⇒ x - y; ELSE y - x ⟧
```

```
BEGIN
x ← f(y, z);
p(x) ⇒ z;
y ← q(r);
p(w) ⇒ q(y);
ELSE q(w);
END
```

5.7.3 Abstract Syntax

```
clause ⇐ S(test: form, consequent: form);
statement ⇐ PTR(form, clause);
statementρ ⇐ R(statement);
compound_form ← statementρ;
```

5.7.4 Evaluator

```
ev_statementρ ←
PROC (r: statementρ) ptr_any;
DECL result: ptr_any;
DECL exit_flag: bool;
result ← NIL;
exit_flag ← FALSE;
FOR i ← 1, ..., length(r) TILL exit_flag DO result ← ev_statement(r[i], exit_flag);
result ENDP;

ev_statement ←
PROC (s: statement, exit_flag: bool BYREF) ptr_any;
mval(s) = form ⇒ eval ∘ val(s);
mval(s) = clause ⇒ ev_clause (val(s), exit_flag);
ENDP;

ev_clause ←
PROC (c: clause, exit_flag: bool BYREF) ptr_any;
not ∘ eval_to_type (c.test, bool) ⇒ NIL;
exit_flag ← TRUE;
eval (c.consequent);
ENDP;
```

5.7.5 Discussion

A compound form is a sequence of statements. It may be constructed either (1) implicitly, in an explicit-procedure (c.f. §5.12.1), or (2) explicitly, and then denoted by BEGIN ... END bracketing.

A compound form is evaluated by evaluating its statements in turn until either (a) the sequence is exhausted, or (b) a statement of type clause occurs in which the test has value TRUE. In the former case, the value of the compound form is the value of the last statement; in the latter case, the value of the compound form is the value of the consequent of the clause.

5.8 ITERATIONS

5.8.1 Concrete Syntax

iteration → FOR identifier ← form , {form ,} ... , form {test} DO form

test → WHILE form | TILL form

5.8.2 Examples

```
FOR i ← 1, ..., n DO sum ← sum + f(i)

FOR i ← 5, 10, ..., k TILL p(i) DO
      x ← x * [[ q(x) ⇒ t(x) ; ELSE t(i) ]]

FOR i ← 1, ..., n DO
   FOR j ← 1, ..., m DO
      f(i , j)
```

5.8.3 Abstract Syntax

```
iteration ⇐ S(index : symbol,
              first : form,
              second : form,
              limit : form,
              test_clause : S(condition : symbol, test : form),
              body : form);
```

Relation of Abstract to Concrete Syntax

Should the second form of an iteration be missing in concrete representation, the field "second" will be NIL in abstract representation. Similarly, a missing test in the concrete program is represented by a NIL value in the field "test_clause.test".

5.8.4 Evaluator

```
ev_iteration ←
PROC(f : iteration) ptr_any;
DECL i, step, limit : int;
DECL cond : bool;
DECL index_address, result : ptr_any;
i ← eval_to_type (f.first, int);
step ← [[ f.second = NIL ⇒ 1; eval_to_type (f.second, int) - i ]];
limit ← eval_to_type (f.limit, int);
NT Since the index variable is local to the iteration, a new variable of
   specified name is created and initialized to the value of i;
index_address ← install_variable (f.index, int, ⟨ ⟩, NIL);
make_current(1);
val(index_address) ← i;
```

```
result ← BEGIN
    NT  There are two routines for performing iteration:  the second
        processes iterations having a test clause, the first processes
        iterations with no test;
    f.test_clause = NIL ⇒
        iterate (index_address, step, limit, f.body, NIL);
    cond ← [[ f.test_clause.condition = "TILL" ⇒ TRUE;
              f.test_clause.condition = "WHILE" ⇒ FALSE ]];
    iterate_with_test(index_address, step, limit, f.body, NIL,
                      cond, f.test_clause.test);
    END;
NT  The index variable created above is deleted;
remove_variables (1);
result ENDP;

iterate ←
PROC(ip : ptr_any, step : int, limit : int, body : form, old_value : ptr_any)
      ptr_any;
(sign(step)*(val(ip) - limit)) > 0 ⇒ old_value;
old_value ← eval(body);
val(ip) ← val(ip) + step;
iterate(ip, step, limit, body, old_value);
ENDP;
```

```
iterate_with_test ←
PROC (ip: ptr_any, step: int, limit: int, body: form, old_value: ptr_any,
      cond: bool, test: form) ptr_any;
DECL saved_flag, test_value: bool;
(sign(step) * (val(ip) - limit)) > 0 ⇒ old_value;
[[ pure_value(old_value) ⇒ [[ old_value ← save(old_value); saved_flag ← TRUE ]] ]];
test_value ← eval_to_type(test, bool);
[[ saved_flag ⇒ old_value ← unsave(old_value) ]];
test_value = cond ⇒ old_value;
old_value ← eval(body);
val(ip) ← val(ip) + step;
iterate_with_test(ip, step, limit, body, old_value, cond, test);
ENDP;
```

5.8.5 Discussion

An iteration specifies the repeated evaluation of some form for changing values of an index. An iteration consists of an index, an iteration-list, a possible test, and an iteration body.

The index is a variable, denoted in concrete representation by an identifier, whose type is INT. The scope of the variable is the iteration; hence, it is created in evaluating the iteration and destroyed when the iteration terminates.

The iteration list consists of either two or three forms. The first and last of these specify initial value and upper limit for the index. If only two forms are in the iteration list, the iteration step is taken by default to be 1. If three forms are present, the step is given by the difference between the values of the second and first forms.

The test, if present, consists of a halting condition (either TILL or

WHILE) and a body. The test body is evaluated just prior to each evaluation of the iteration body. If the value of the test body agrees with the halting condition, then the iteration ceases at that point (without again evaluating the iteration body).

5.9 MODES

5.9.1 Concrete Syntax

mode_constant → INT | BOOL | CHAR | NONE | NONEREF | PTR_ANY | STACK

m_form → row_form | struct_form | ptr_form | rany_form | m_def

row_form → row_symbol ({form , } form)

row_symbol → ROW | R

struct_form → struct_symbol ({ identifier : form , }$^{\circledast}$)

struct_symbol → STRUCT | S

ptr_form → PTR({form , }$^{\oplus}$)

rany_form → RANY({form , }$^{\circledast}$)

m_def → identifier ⇐ form

5.9.2 Examples

ROW (3 , INT)

R(⟦ x < y ⇒ 10 ; 2 * n ⟧ , ⟦ p(x) ⇒ INT ; BOOL ⟧)

R(R(complex))

STRUCT (re : INT, im : INT)

S(amp_rating : INT, manufacturer : R(n, CHAR))

PTR(bool_matrix)

PTR(CHAR, BOOL, ⟦ n > 0 ⇒ INT ; complex ⟧)

RANY(INT, complex)

RANY(BOOL, CHAR, PTR(string, intρ))

```
triple ⇐ R(3, INT)

complex ⇐ pair ⇐ S(re : INT, im : INT)

header ⇐ PTR(triple, pair, INT)

int_or_bool ⇐ RANY(INT, BOOL)
```

5.9.3 Abstract Syntax

```
m_form ⇐ PTR(row_form, struct_form, ptr_form, rany_form, m_def);

row_form ⇐ S(length : form, type : form);

struct_form ⇐ R(struct_component ⇐ S(name : symbol, type : form));

ptr_form ⇐ R(form);

rany_form ⇐ R(form);

m_def ⇐ S(name : symbol, type : form);
```

Relation of Abstract to Concrete Syntax

A concrete row-form whose first form is missing has an abstract row-form with "length" field NIL.

5.9.4 Auxiliary Mode Definitions

```
mode ⇐ PTR (ddb);

modeρ ⇐ R(mode);

type_descriptor ⇐ PTR(row_def, struct_def, modeρ);

ddb ⇐ S(d : type_descriptor,
        class : symbol,
        type_resolved : bool,
        dope_length : int,
        a_fn : proc_var,
        s_fn : proc_var,
        canonical_name : symbol);
```

```
row_def ⇐ S(element_type : mode, length : int);
struct_def ⇐ R(struct_def_component ⇐
                    S(field_name : symbol, field_type : mode));
```

5.9.5 Evaluator

```
ev_mform ←
PROC(f : m_form) ptr_any;
DECL m : mode;
DECL result : ptr_any;
m ← BEGIN
    NT There are five classes of m-forms, each being handled by a
       separate evaluator. In each case, the evaluator returns a mode,
       i.e., a pointer to a ddb in which the data-type definition resides;
    mval(f) = row_form ⇒ ev_row ∘ val(f);
    mval(f) = struct_form ⇒ ev_struct ∘ val(f);
    mval(f) = ptr_form ⇒ ev_ptr ∘ val(f);
    mval(f) = rany_form ⇒ ev_rany ∘ val(f);
    mval(f) = m_def ⇒ ev_mdef ∘ val(f);
    END;
NT The value m is "returned" by copying it into the result-slot and
    returning a pointer to the copy;
free_last (result_slot);
result ← get_stack_space (result_slot, mode, ⟨ ⟩);
val(result) ← m;
result;
ENDP;
```

```
ev_row ←
PROC (r : row_form) mode ;
DECL m, m_elt : mode ;
DECL n : int ;
NT  Storage for the created ddb is obtained from the heap ;
m ← allocate (ddb, ⟨ ⟩) ;
m.class ← "row" ;
m.type_resolved ← TRUE ;
m.d ← allocate (row_def, ⟨ ⟩) ;
NT  m_elt is the mode of the elements constituting the row ;
m_elt ← eval_to_type (r.type, mode) ;
m_elt.type_resolved = FALSE ⇒ error ("row_error") ;
m.d.element_type ← m_elt ;
NT  n contains either (1) the number of elements constituting a row of this
    mode, if this number is fixed, or (2) the code -1, if this is not fixed ;
n ← ⟦ r.length = NIL ⇒ -1 ;
      ELSE  eval_to_type (r.length, int) ⟧ ;
m.d.length ← n ;
NT  The "dope-length" field contains the number of length unresolved
    dimensions ;
m.dope_length ← m_elt.dope_length + ⟦ n = -1 ⇒ 1 ; 0 ⟧ ;
NT  The assignment and selection functions for this mode are constructed
    by two primitives ;
m.a_fn ← build_row_assigner (m_elt, n) ;
m.s_fn ← build_row_selector (m_elt, n) ;
m ENDP ;
```

```
ev_struct ←
PROC (s: struct_form) mode;
DECL m, m_elt: mode;
DECL n: int;
DECL r: ptr_any;
n ← length (s);
m ← allocate (ddb, ⟨⟩);
m.class ← "struct";
m.type_resolved ← TRUE;
m.d ← allocate (struct_def, ⟨n⟩);
m.dope_length ← 0;
FOR i ← 1, . . . , n DO
    BEGIN
    m.d[i].field_name ← s[i].name;
    m_elt ← eval_to_type (s[i].type, mode);
    m_elt.type_resolved = FALSE ⇒ error ("struct_error");
    m.d[i].field_type ← m_elt;
    m.dope_length ← m.dope_length + m.elt.dope_length;
    END;
m.a_fn ← build_struct_assigner (m.d);
m.s_fn ← build_struct_selector (m.d);
m ENDP;
```

```
ev_ptr ←
PROC (f: ptr_form) mode;
DECL n: int;
DECL m: mode;
n ← length(f);
m ← allocate (ddb, ⟨ ⟩ );
m.class ← "ptr";
m.type_resolved ← TRUE;
m.d ← allocate ( modeρ, ⟨n⟩ );
FOR i ← 1, . . . , n DO
    [[ m . d[i] ← eval_to_type(f[i], mode);
      m . d[i] = NIL ⇒ error("ev_ptr_error") ]]
m.dope_length ← 0;
m.s_fn ← NIL; NT can't select a component of a pointer;
m.a_fn ← BEGIN
    n = 1 ⇒ build_ptr_assigner(m.d[1]);
    ELSE build_united_ptr_assigner (m.d);
    END;
m ENDP;
```

```
ev_rany ←
PROC(r : rany_form) mode;
DECL n : int;
DECL m : mode;
n ← length (r);
m ← allocate (ddb, ⟨ ⟩);
m.class ← "rany";
m.type_resolved ← FALSE; NT This is redundant;
m.d ← allocate (modeρ, ⟨n⟩);
NT The descriptor for a rany is a row of the alternative modes. Each of
    these must be resolved;
FOR i ← 1, . . . , n DO
    BEGIN
    m.d[i] ← eval_to_type (r[i], mode);
    m.d[i].type_resolved = FALSE ⇒ error ("rany_error");
    END;
m ENDP;
```

```
ev_mdef ←
PROC (d : m_def) mode ;
DECL p : ptr_any ;
DECL ml, mr : mode ;
p ← eval_symbol (d.name) ;
NT  The "name"field of an m_def must be a mode_valued variable ;
mval(p) ≠ mode ⇒ error ("mdef_ error") ;
NT  If the mode valued variable does not already point to a ddb, it is made
    to do so ;
[[ val(p) = NIL ⇒ val(p) ← allocate (ddb, ⟨ ⟩) ]] ;
ml ← val(p) ;
mr ← eval_to_type (d.type, mode) ;
val(ml) ← val(mr) ;
ml . canonical_name ← d.name ;
NT  The value of an m_def is the value of its "type" field ;
mr ENDP ;
```

5.9.6 Discussion

A mode corresponds to the intuitive notion of data type. It specifies the structure and other properties of a set of values which are said to be of that mode, or have that mode. Modes are themselves values and may be created, compared, and manipulated.

Mode values originate in EL1 as either (1) the value of a mode constant, or (2) the value of an m-form. In either case, a mode is a pointer to a ddb (stored in the heap) which defines a data type.

A ddb for a data type $\mathscr{D}$ consists of the following:

(1) the class of $\mathscr{D}$ (row, struct, ptr or rany),

(2) a flag specifying whether or not $\mathscr{D}$ is type resolved,

(3) the number of length unresolved dimensions (dope-length),

(4) a canonical name which, when present, is an external denotation for the ddb,

(5) two proc-vars, each pointing to a procedure – one for assigning to and one for selecting from an object of type $\mathscr{D}$,

(6) a type-descriptor, which specifies the structure of an object of type $\mathscr{D}$.

Mode constants are primitives in the language. The value, $\mathfrak{M}^*$, of a mode constant, $\mathfrak{M}$, is a mode which points to a fixed, predefined ddb. As with all constants, the value $\mathfrak{M}^*$ is a pure value; i.e., $\mathfrak{M}^*$ appears in the result slot.

The four mode valued operators ROW, STRUCT, PTR, and RANY create a new ddb on each call and assign appropriate values to the ddb's fields. Modes produced by these operators are said to belong to class row, struct, ptr, and rany, respectively; objects having such modes are referred to as rows, structs, and ptrs, respectively.‡

A row is a compound object whose components have identical mode and size. The number of components in a row is obtained by applying the function "length" to the row. Any of the components may be selected by subscripting the object with a single form having INT value (c.f. §5.10). There are two acceptable formats for the arguments to ROW, producing two sorts of modes (both of class row).

(1) If two operands are present, the value of the first (which must be an INT) is the number of components in the row being defined.

---

‡ The remaining case (ranys) never occurs as it is impossible to create an object whose mode is of class rany.

(2) If the first operand is omitted, then the number of components is unspecified and the mode is said to be length unresolved.

A struct is a compound object whose components may have differing modes and sizes. The number of components and the mode of each component are fixed for all structs of a given mode. Selection of a component from a struct is performed by the operation of selection (c.f. §5.10) and may use either an INT index or a symbolic name as a designator.

The attribute length unresolved may apply to modes of class struct as well as those of class row. In the latter case, this may be either because the number of components is undetermined or because the mode of the components is itself length unresolved. A struct mode will be length unresolved if any of its components is. Each length unresolved mode may be assigned an integer greater than or equal to 1 (termed a dope-length‡) defined to be the number of distinct lengths which must be specified for that mode to be fully resolved. The dope-length of a struct is the sum of dope-lengths of its component modes. The dope-length of a row is either (a) the dope-length of the mode of its components, or (b) this number plus one, depending on whether or not the number of components is fixed. In creating a row or struct whose mode is length unresolved, it is necessary to supply a dope-vector of dope-length INT's; the dope-vector is said to resolve the mode (c.f. §5.13.4).

The operator PTR, like ROW, delivers modes of two subclasses, depending on the format of its argument list. Given a single argument, PTR delivers a simple pointer mode; given two or more arguments, PTR

‡ A dope-length may also have value zero, indicating that the associated mode is fully length resolved.

delivers a united pointer mode. These subclasses differ in the following fashion. Assume that all arguments to PTR deliver a value of type mode (otherwise an error will occur) and denote these values by $\mathfrak{M}_i$. If p has mode PTR($\mathfrak{M}_1$), then p is a simple pointer and can be assigned to point to values having mode $\mathfrak{M}_1$. Subsequent to such assignment, val(p) is the value to which p points. On the other hand, if p' has mode PTR($\mathfrak{M}_1, \ldots, \mathfrak{M}_n$), with $n \geq 2$, then p' is a united pointer and can be assigned to point to values having modes $\mathfrak{M}_1$, or $\mathfrak{M}_2$, or ..., or $\mathfrak{M}_n$. The form val(p') yields the value to which p' points.

The class rany differs from the other three classes in that no value has mode of this class. (Modes of this class can be created, but not corresponding values.) RANY takes as arguments n forms with mode values $\mathfrak{M}_1, \ldots, \mathfrak{M}_n$ and produces a mode $\mathfrak{M}'$ which is said to be type unresolved with alternatives $\mathfrak{M}_1, \ldots, \mathfrak{M}_n$. It is possible to declare a formal parameter or declared variable (c.f. §5.12.1) to have mode $\mathfrak{M}'$, the interpretation being that the variable so declared is either an $\mathfrak{M}_1$, or an $\mathfrak{M}_2$, or ..., or an $\mathfrak{M}_n$. Each time such a variable is created, $\mathfrak{M}'$ must be type resolved by specifying which of the alternative modes is to be the actual mode of that instance of the variable (c.f. §5.13.4).

The remaining m-form is m-def, written as a binary operation in the format "$\mathcal{S} \Leftarrow \mathcal{F}$" where $\mathcal{S}$ is an identifier and $\mathcal{F}$ is a form. $\mathcal{S}$ must denote a mode valued variable, $\mathfrak{M}$, and $\mathcal{F}$ must evaluate to a mode, $\mathfrak{M}'$. Evaluation of the m-def establishes a two-way relation between $\mathcal{S}$ and $\mathfrak{M}'$, as follows. (1) The ddb to which $\mathfrak{M}$ points is given the same value as the ddb to which $\mathfrak{M}'$ points. (If $\mathfrak{M}$ has value NIL, it is set to point to a newly allocated ddb and m-def proceeds as above.) (2) The "canonical-name" field in val($\mathfrak{M}$) is assigned $\mathcal{S}$ as its value. As a result of these two actions,

the mode $\mathfrak{M}$, denoted by $\mathscr{S}$, points to a ddb with value equal to val($\mathfrak{M}'$), while this ddb has a field which refers back to $\mathscr{S}$. The identifier $\mathscr{S}$ thus serves as an external representation of the otherwise non-denotable ddb; further, this external representation can be recovered from the ddb itself.

5.10 SELECTION

5.10.1 Concrete Syntax

selection → form2 field

field → . identifier | [form]

5.10.2 Examples

a[i]

a[f(x) + 3 * a[i]]

z . re

time_card . name . first_initial

positions [d(t)] . im

5.10.3 Abstract Syntax

selection ⇐ S(object : form, field : form) ;

Relation of Abstract to Concrete Syntax

The concrete format for selectors

form . identifier (e.g., name . first_initial)

is syntactic sugar for the equivalent concrete format

form ["identifier"] (e.g., name ["first_initial"]) .

The conversion from concrete (external) representation to internal representation replaces instances of the former type by equivalent forms of the

latter type. Hence, the abstract representation has provision only for the latter.

5.10.4 Evaluator

```
ev_selection ←
PROC (s: selection) ptr_any;
DECL x, result: ptr_any;
DECL saved_flag: bool;
saved_flag ← FALSE;
x ← dereference ∘ eval (s.object);
[[ pure_value(x) ⇒ BEGIN x ← save(x); saved_flag ← TRUE END ]];
result ← select2(x, field_index(mval(x), s.field));
[[ saved_flag ⇒ result ← unsave(result) ]];
result ENDP;

dereference ←
PROC (x: ptr_any) ptr_any;
mval(x).class ≠ "ptr" ⇒ x;
ELSE dereference ∘ val(x);
ENDP;

unsave ←
PROC(p: ptr_any) ptr_any;
p ← return_result(p, mval(p));
free_last(value_stack);
p ENDP;
```

```
save ←
PROC (in: ptr_any) ptr_any;
DECL out: ptr_any;
out ← get_stack_space (value_stack, mval(in), dope_vector(in));
assign2 (out, in);
out ENDP;

field_index ←
PROC (m: mode, f: form) int;
NT This evaluates the field of a selection and produces an int index;
DECL name: symbol;
DECL p: ptr_any;
p ← eval(f);
m.class = "row" ⇒
    BEGIN
    mval(p) = int ⇒ val(p);
    ELSE error("field_error");
    END;
m.class = "struct" ⇒
    BEGIN
    mval(p) = int ⇒ val(p);
    mval(p) ≠ symbol ⇒ error("field_error");
    name ← val(p);
    index ← 0;
    FOR i ← 1, ..., length (m.d) TILL index ≠ 0 DO
        [[ m.d[i].field_name = name ⇒ index ← i ]];
    index ≠ 0 ⇒ index;
    ELSE error ("field_error");
    END;
```

```
ELSE error("field_error");
ENDP;
```

5.10.5 Discussion

A selection designates a single component from a compound object, that is, from a row or a struct. A selection is written as a form followed by a field. Since a selection is a form, the operation of selection may be repeated (i.e., form $field_1$ $field_2$ . . . $field_n$) and associates to the left. Fields are denoted in two ways: (1) by a dot followed by an identifier which is treated as a constant symbol, (2) by a bracketed form which is evaluated. The former format is syntactic sugar for a special case of the latter format.

Evaluation of a selection begins with evaluation of its form. If this value is a pointer, it is dereferenced; i.e., the operator val (c.f. §5.15) is repeatedly applied until a non-pointer value is obtained. If the resulting value is a pure value, it is saved. Having thus obtained a value $\mathscr{V}$, the evaluator next evaluates the field. If its value is a symbol, the symbol is converted to an integer index; if its value is an int, the int is used directly as an index. The operation of selection is carried out by applying the selection procedure for the mode of $\mathscr{V}$ to $\mathscr{V}$ and the index $\mathscr{S}$, obtaining the $\mathscr{S}^{th}$ component of $\mathscr{V}$, $\mathscr{V}_{\mathscr{S}}$. As with all values in the language, $\mathscr{V}_{\mathscr{S}}$ is represented‡ in such fashion that assignments to $\mathscr{V}_{\mathscr{S}}$ are well-defined and may change the value of $\mathscr{V}$.

‡ Such representation is referred to as an L-value in CPL [CPL 66].

5.11 AGGREGATES

5.11.1 Concrete Syntax

aggregate → ⟨ {form :} {SIZE form :} {form , }* ⟩

5.11.2 Examples

⟨ 2, 3, 2+3, 11*(7+4) ⟩

⟨ complex : 2*x, y ⟩

⟨ string : 's, 'n, 'o, 'b, 'o, '1 ⟩

5.11.3 Abstract Syntax

aggregate ⇐ S (type : form, size : form, components : formρ) ;

5.11.4 Evaluator

```
ev_aggregate ←
PROC (a : aggregate) ptr_any ;
DECL n : int ;
DECL p : ptr_any ;
DECL m : mode ;
n ← length (a . components) ;
p ← NT  This gets space on the value stack to hold the compound object ;
    BEGIN
    a . type = NIL ⇒ get_stack_space (value_stack, intρ, ⟨ n ⟩) ;
    m ← eval_to_type (a . type, mode) ;
    a . size ≠ NIL ⇒
        get_stack_space (value_stack, m, eval_to_type (a . size, intρ)) ;
    m . dope_length = 1 ⇒ get_stack_space (value_stack, m, ⟨ n ⟩) ;
    ELSE  error ("aggregate_error") ;
    END ;
```

```
FOR i ← 1, . . . , n DO
    NT This evaluates the components of the aggregate and fills in the
        components of the compound object;
    assign 2 (select2(p, i), eval(a . components[i]));
NT The compound object is copied into the result slot and the space
    allocated above on the value stack is freed;
p ← return_result (p, mval(p));
free_last (value_stack);
p ENDP;
```

5.11.5 Discussion

An aggregate creates and gives value to a compound object. It is a convenient and efficient means for creating compound values such as vectors, complex numbers, and payroll records.

An aggregate consists of a type, a size, and zero or more components. If either the type or size is missing, the default values are intρ and the number of components, respectively. The value of an aggregate is a compound object of specified type and size, having components equal to the value of the components of the aggregate. The value of an aggregate is a pure value; hence, it is left in the result-slot.

5.12 PROCEDURES

5.12.1 Concrete Syntax

proc_constant → explicit_procedure | code_procedure

formal_parameter → identifier : type {bind_class}

type → mode_constant | identifier

bind_class → BYREF | BYVALUE | UNEVALED

declaration → DECL {identifier , }$^{\oplus}$: type options

options → {SPECIF form} {SIZE form}

explicit_procedure → PROC ({formal_parameter , }$^{\circledast}$) type ;
{declaration ;}* {statement ;}$^{\circledast}$ ENDP

Note

(1) Code-procedure is an undefined nonterminal (c.f. §5.12.6).

(2) If the bind-class of a formal-parameter is omitted, it is taken by default to be BYREF.

Cross Reference

form	–	§5.3.1
identifier	–	§5.5.1
mode-constant	–	§5.9.1
statement	–	§5.7.1

5.12.2 Examples

(a) Spur of a square matrix of integers:

```
PROC (a : int_matrix) int ;
DECL  s : int ;
FOR  k ← 1, ..., length(a)  DO  s ← s + a[k][k] ;
ENDP
```

(b) Transpose of a square matrix of integers:

```
PROC (b: int_matrix) none;
NT This procedure transposes the matrix b in place;
DECL n, temp: int;
FOR i ← 1, ..., n ← length(b) DO
    FOR k ← i+1, ..., n DO
        ⟦ temp ← b[i][k]; b[i][k] ← b[k][i]; b[k][i] ← temp ⟧;
ENDP
```

5.12.3 Abstract Syntax

```
proc_var ⇐ PTR (explicit_procedure, code_procedure);
expr_formal ⇐ S (name: symbol, type: form, bind_class: symbol);
expr_formalρ ⇐ R (expr_formal);
declaration ⇐ S (names: symbolρ ⇐ R(symbol),
                 type: form,
                 specif: form,
                 size: form );
declarationρ ⇐ R(declaration);
explicit_ procedure ⇐ S (formals: expr_formalρ,
                         result_type: form,
                         declarations: declarationρ,
                         statements: statementρ);
```

Relation of Abstract to Concrete Syntax

(1) Although a type in the concrete syntax is restricted to be either a mode-constant or an identifier, it is represented in the abstract syntax as a form (c.f. expr-formal, declaration, and explicit-procedure).

(2) The field "bind-class" of an abstract expr-formal is the bind-class specified in the concrete formal-parameter, rendered as a symbol (e.g. "BYVALUE").

(3) The two options of a concrete declaration are represented by two forms in an abstract declaration: a missing option has a NIL form.

5.12.4 Auxiliary Mode Definitions

```
code_formal ⇐ S(name: symbol, type: mode, bind_class: symbol);
code_formalρ ⇐ R(code_formal);
machine_code ⇐ R(int);
code_procedure ⇐ S(formals: code_formalρ,
                   result_type: form,
                   body: machine_code);
formal_parameterρ ⇐ RANY (expr_formalρ, code_formalρ);
procedure_block ⇐ RANY (explicit_procedure, code_procedure);
```

5.12.5 Evaluator

For the value of a proc-constant, see section 5.4.4; for the application of a procedure to its arguments, see section 5.13.4.

5.12.6 Discussion

Procedure values are commonly created as proc-constants, of which there are two types. An explicit-procedure defines a process by EL1 text and is written in the format "PROC . . . ENDP". A code-procedure is written in some language other than EL1, typically machine language. While there are certain builtin code-procedures, no external representation is defined in the reference language. It is assumed that implementations will choose some representation (e.g., symbolic assembly language) suitable to their background machine.

Both types of proc-constants are represented internally by proc-vars (c.f. §5.12.3), i.e. pointers to the locations in which the actual defining bodies reside. In the case of explicit procedures, the defining body is created, in heap storage, at the time the external representation (source text) is translated to internal representation.

The abstract syntaxes of explicit and code procedures are closely related. Both have a return-mode, i.e. a form‡ whose value is the data type of the result delivered by the procedure. Also, both have a set of formals which specify the internal name, data type, and method of binding for each argument position. One difference should be noted: in code-procedures the data type of a formal is specified by a fixed mode, whereas in explicit procedures the type of a formal is specified by a form‡ to be evaluated when the procedure is entered (c.f. §5.13.4).

---

‡Note, however, that the concrete syntax (c.f. §5.12.1) restricts such a form to be either a mode-constant or an identifier.

5.13 PROCEDURE APPLICATION

5.13.1 Concrete Syntax

procedure application → form (arguments)

arguments → {form , }$^{\circledast}$

5.13.2 Examples

```
f(x)

g (x, y ← h(w, z), [[ p(x) ⇒ q(x) ; (a * w) + b ]])

[[ p(w) ⇒ foo ; ELSE fum ]] (y + z)

(revert (s, t)) (x, y, z)

PROC (x: int, y: int) int ; x > y ⇒ -y ; -x ENDP (f(a), a + x * f(b))
```

5.13.3 Abstract Syntax

formρ ⇐ R (form)

procedure_application ⇐ S(operator : form, arguments : formρ)

5.13.4 Evaluator

```
apply ←
PROC (f : procedure_application) ptr_any ;
apply2 (checkproc ∘ eval(f . operator), f . arguments) ;
ENDP ;
```

```
apply2 ←
PROC (p: procedure_block, args: formρ) ptr_any;
DECL old_pdl_index: int;
DECL declared_type: mode;
DECL result: ptr_any;
old_pdl_index ← pdl_index; NT pdl_index is free;
bind_formals (p.formals, args);
make_current(length(args));
NT The result_type is evaluated in an environment which includes bound
    values of formal parameters;
declared_type ← eval_to_type (p.result_type, mode);
[[ typ(p) = code_procedure ⇒ result ← xct(p.body, name_pdl, pdl_index);
    NT Otherwise, p is an explicit_procedure;
    ev_declarations(p.declarations);
    result ← ev_statementρ (p.statements) ]];
result ← proc_exit (result, declared_type, old_pdl_index);
pdl_index ← old_pdl_index;
result ENDP;

bind_formals ←
PROC(p: formal_parameterρ, args: formρ) none;
DECL m: mode;
DECL arg, new: ptr_any;
length(p) ≠ length(args) ⇒ error("binding_error")
FOR i ← 1, ..., length(args) DO
    BEGIN
    m ← [[ typ(p[i]) = code_formal ⇒ p[i].type;
          ELSE eval_to_type (p[i].type, mode) ]];
```

```
NT  There are three cases:  (1) passing the argument unevaluated,
    (2) passing the evaluated argument by reference, (3) passing the
    evaluated argument by value;
    p[i].bind_class = "UNEVALED" ⇒
        BEGIN
        m ≠ form ⇒ error("binding_error");
        new ← install_variable (p[i].name, form, ⟨⟩, NIL);
        val(new) ← args[i];
        END;
    arg ← eval(args[i]);
    [[ m.class = "rany" ⇒ m ← resolve(m, mval(arg));
      not ∘ compatible (m, mval(arg)) ⇒ error ("binding_error") ]];
    (p[i].bind_class = "BYREF") ∧ not ∘ pure_value(arg) ⇒
          install_variable (p[i].name, NIL, ⟨⟩, arg);
    new ← install_variable (p[i].name, m, dope_vector(arg), NIL);
    assign2(new, arg);
    END;
ENDP;

ev_declarations ←
PROC (d: declarationρ) none;
DECL m: mode;
FOR i ← 1, . . ., length(d) DO
    BEGIN
    m ← eval_ to_type(d[i].type, mode);
    [[ m.class = "rany" ⇒
```

```
        BEGIN
        d[i].specif = NIL ⇒ error("decl_error");
        m ← resolve(m, eval_to_type(d[i].specif, mode));
        END ][;
    FOR j ← 1, ..., length(d[i].names) DO
        install_variable(d[i].names[j], m,
                         [[ d[i].size = NIL ⇒ ⟨ ⟩;
                            eval_to_type(d[i].size, intρ) ]], NIL);
    END;
ENDP;

proc_exit ←
PROC (result: ptr_any, expected_mode: mode, old_pdl_index: int) ptr_any;
[[ expected_mode = none ⇒ result ← NIL;
    expected_mode.class = "rany" ⇒
        expected_mode ← resolve(expected_mode, mval(result));
    not ∘ compatible (expected_mode, mval ∘ result) ⇒
        error("proc_exit_error") ]];
FOR i ← pdl_index, pdl_index - 1, ..., old_pdl_index + 1 DO
    BEGIN
    [[ last_in(result, value_stack) ⇒
        result ← return_result(result, expected_mode) ]];
    remove_variables (1);
    END;
mval(result) = expected_mode ⇒ result;
return_result(result, expected_mode);
ENDP;
```

```
resolve ←
PROC (u: mode, r: mode) mode;
NT resolve(u, r) returns r if r is one of the alternatives of u, otherwise
    resolve calls error;
DECL found_flag: bool;
u.class ≠ "rany" ⇒ error("resolve_error");
FOR i ← 1, ..., length(u.d) TILL found_flag DO
    [[ u.d[i] = r ⇒ found_flag ← TRUE ]];
found_flag = TRUE ⇒ r;
ELSE error("resolve_error");
ENDP;
```

Cross Reference

assign2	–	§5.14
checkproc	–	§5.6.4
ev-statementρ	–	§5.7.4
install-variable	–	§5.14
remove-variable	–	§5.14

5.13.5 Discussion

A procedure application serves to invoke the application of its operator, a procedure, to its arguments. This is carried out in five steps:

(1) evaluation of the operator to obtain a procedure,

(2) binding the formal parameters to the arguments,

(3) evaluation of the result-type to obtain the declared type of the procedure,

(4) evaluation of the procedure body,

(5) exiting the procedure.

The operator may be any form which evaluates to an explicit-procedure (c.f. §5.12.3), a code-procedure (c.f. §5.12.4), or a pointer to either of these. In particular, a procedure-application which delivers a procedure is a legal operator.

The formal-parameters are bound in lexographical order from left to right; each parameter is bound in the environment of variables which existed when the procedure application began. That is, the binding of a variable to an argument does not change the set of variables seen by the evaluation of arguments to its right. A parameter can be bound in one of three ways: UNEVALED, BYVALUE, or BYREF. In the first case, the argument is not evaluated; the formal parameter, which must have mode form, is bound to the unevaluated argument. In the other two cases, the argument is evaluated, producing a value $\mathcal{O}$. Binding BYVALUE creates a new object $\mathcal{O}'$ (having the same mode and size) and initializes $\mathcal{O}' \leftarrow \mathcal{O}$. Binding BYREF creates no new object but rather establishes a temporary connection between $\mathcal{O}$ and the name $\mathcal{S}$ of the formal-parameter, so that use of the name $\mathcal{S}$ yields $\mathcal{O}$.

The value of the field "result-type" is said to be the declared type of the procedure, i.e. the expected mode of its result. The mode of its actual result must be equal to or compatible with this.

Evaluation of the procedure body takes place in two possible ways, depending on the type of procedure. If it is a code-procedure, the body is executed. If it is an explicit-procedure, its declarations are first evaluated and its statements component is evaluated as a compound form. In either case, the procedure delivers some value $\mathcal{O}$.

Exiting the procedure entails destroying the new objects created by parameter binding and declaration evaluation. If the procedure value $\mathcal{O}$ is an object to be so destroyed (or a component of such an object), then a new object $\mathcal{O}'$ is created and initialized $\mathcal{O}' \leftarrow \mathcal{O}$. $\mathcal{O}'$ becomes the procedure value and $\mathcal{O}$ is destroyed.

The final procedure value $\mathcal{O}''$ is checked against the declared type. If the type is correct, no further action is taken; if $\mathcal{O}''$ can be converted to the proper type, the conversion is performed; otherwise, an error occurs.

5.14 AUXILIARY ROUTINES USED BY THE EVALUATORS

Routines are listed in alphabetical order.

```
assign2 ←
PROC (left: ptr_any, right: ptr_any) ptr_any;
DECL ml, mr: mode;
DECL p: proc_var;
ml ← mval(left); mr ← mval(right);
not ∘ compatible(ml, mr) ⇒ error("assign_error");
p ← ml.a_fn;
install_variable(p.formal_parm[1].name, NIL, ⟨⟩, left);
install_variable(p.formal_parm[2].name, NIL, ⟨⟩, right);
make_current(2);
x.ct(p.body, name_pdl, pdl_index);
remove_variables (2);
right ENDP;
```

```
compatible ←
PROC (sink: mode, source: mode) bool;
```

NT compatible(sink, source) = TRUE iff a value of type source can be assigned to a value of type sink. Four cases are admissible;

```
DECL flag: bool;
sink = source ⇒ TRUE;
(sink.class = "ptr") ∧ (source = NONEREF) ⇒ TRUE;
(sink =  ptr_any) ∧ (source.class = "ptr") ⇒ TRUE;
(sink.class = "ptr") ∧ (source.class = "ptr") ∧ (length(source.d) = 1) ⇒
    BEGIN
    flag ← FALSE;
    FOR i ← 1, ..., length (sink.d) TILL flag DO
        [[ sink.d[i] = source.d[1] ⇒ flag ← TRUE ]];
    flag;
    END;
ELSE FALSE;
ENDP;

dope_vector ←
PROC(p: ptr_any) intρ;
NT  dope_vector creates a dope vector for the object to which p points;
DECL  v: intρ SIZE ⟨ mval(p).dope_length ⟩;
FOR i ← 1, ..., length(v) DO
    v[i] ← length2(p, i);
v  ENDP;

error ←
PROC(s: symbol) ptr_any;
NT  error searches for a procedure named s. If one is found, it is
    evaluated. If not, an error message is printed and error 2 is
    called to handle the error;
```

```
DECL p: ptr_any;
p ← val(s).datum;
(mval(p) = explicit_procedure) ∨ (mval(p) = code_procedure) ⇒
    apply 2 (val(p), ⟨formρ:⟩);
print ("ERROR: ");
print (s);
error2( );
ENDP;

eval_to_type ←
PROC (f: form, m: mode) m;
NT  The mode of the result delivered by this procedure is equal to the second
    argument;
DECL p: ptr_any;
p ← eval(f);
mval(p) = m ⇒ val(p);
ELSE error ("eval_to_type_error");
ENDP;
```

```
initialize ←
PROC (p : ptr_any) none ;
NT  initialize (p) assigns an initial value to the object which p points to ;
DECL  m : mode ;
m  ←  mval(p) ;
m  =  int  ⇒  val(p)  ←  0 ;
m  =  bool  ⇒  val(p)  ←  FALSE ;
m = char  ⇒  val(p)  ←  ' ' ;
m.class  =  "ptr"  ⇒  val(p)  ←  NIL ;
m.class  =  "row"  ⇒
    FOR  i ← 1, ..., length(p)  DO  initialize ∘ select2(p, i) ;
m.class  =  "struct"  ⇒
    FOR  i ← 1, ..., length(m.d)  DO  initialize ∘ select2(p, i) ;
NT  otherwise, p points to a STACK which requires special initialization ;
ENDP ;

install_variable ←
PROC (name : symbol, type : mode, dope_vector : intρ, value : ptr_any) ptr_any ;
NT  install_variable creates a variable whose name is specified by the first
    argument.  This entails (1) adding an entry on the name_pdl, and
    (2) possibly obtaining new storage on the value_stack.
```

```
pdl_index = pdl_index_max ⇒ error("pdl_overflow");
pdl_index ← pdl_index + 1;
name_pdl[pdl_index].name ← name;
name_pdl[pdl_index].datum ←
    BEGIN
    value ≠ NIL ⇒ value;
    get_stack_space (value_stack, type, dope_vector);
    END;
ENDP;

make_current ←
PROC(n) none;
NT  This makes the top n elements of the name_pdl to be current
    bindings;
DECL name: symbol;
FOR  i ← name_pdl, name_pdl - 1, ..., (name_pdl - n) + 1  DO
    BEGIN
    name ← name_pdl[i].name;
    name_pdl[i]. old_index ← name.pdl_position;
    name.pdl_position ← i;
    name.datum ← name_pdl[i].datum;
    END;
ENDP;
```

```
pure_value ←
PROC (p: ptr_any) bool;
last_in (p, result_slot);
ENDP;

remove_variables ←
PROC (n: int) none;
NT  This removes variables from the name_pdl and, where the binding is
    not BYREF, from the value_stack as well;
DECL s: symbol;
DECL k: int;
FOR i ← 1, . . . , n DO
    BEGIN
    [[ last_in(name_pdl[pdl_index].datum, value_stack) ⇒
                free_last (value_stack)]];
    name_pdl[pdl_index].datum ← NIL;
    s ← name_pdl[pdl_index].name;
    k ← val(s).pdl_position ← name_pdl[pdl_index].old_index;
    val(s).datum ← [[ k > 0 ⇒ name_pdl[k].datum; ELSE NIL ]];
    pdl_index ← pdl_index - 1;
    END;
ENDP;
```

```
return_result ←
PROC (r : ptr_any, m : mode) ptr_any ;
NT This returns the value pointed to by r by copying it into the result-slot.
    If its mode differs from m, it is converted to an m during the copying ;
DECL temp : ptr_any ;
[[ last_in (r, result_slot) ⇒
    BEGIN
    temp ← result_slot ;
    result_slot ← aux_result_slot ;
    aux_result_slot ← temp ;
    END ]] ;
free_last (result_slot) ;
temp ← get_stack_space (result_slot, m, dope_vector (r)) ;
assign2 (temp, r) ;
temp ;
ENDP ;
```

```
select2 ←
PROC(x: ptr_any, i: int) ptr_any;
NT  this returns a pointer to the i-th element of the compound object
    to which x points;
DECL m: mode;
DECL p: proc_var;
DECL temp: ptr_any;
m ← mval(x);
p ← m.s_fn;
install_variable (p.formal_parm[1].name, NIL, ⟨ ⟩, x);
temp ← install_variable (p.formal_parm[2].name, int, ⟨ ⟩, NIL);
val(temp) ← i; make_current(2);
temp ← xct(p.body, name_pdl, pdl_index);
remove_variables(2);
temp ENDP;
```

5.15 PRIMITIVE PROCEDURES

A procedure is primitive if it exists in the language but cannot be acceptably defined in the language. Either it is so elementary as to admit no definition in terms of simpler notions or it is such that any definition would involve machine-dependent concepts. In either case, for the purpose of formal definition it is a primitive.

The following listing groups primitive procedures by function. For each procedure the modes of its arguments and result is specified by a pseudo procedure heading; its operation is described in English. Where appropriate, citations are made to more complete descriptions elsewhere in this paper.

5.15.1 Equality Tests

```
equal_int ~ PROC(x: int, y: int) bool;
equal_bool ~ PROC(x: bool, y: bool) bool;
equal_char ~ PROC(x: char, y: char) bool;
equal_ptr ~ PROC(x: ptr_any, y: ptr_any) bool;
```

Each procedure returns TRUE iff x is equal to y; i.e., identical bit patterns – the number of bits being fixed for each procedure in any given implementation.

5.15.2 Arithmetic Operations

```
> ~
PROC(x: int, y: int) bool;
```

Returns TRUE iff x is greater than y.

```
+, -, *, /  ~
PROC(x: int, y: int) int;
```

The four procedures +, -, *, / have conventional meaning (addition, subtraction, multiplication, division) over the integers.

5.15.3 Pointer Operations

```
val ~
PROC(p: ptr_any) mval(p);†
```

val(p) is the object to which p points.

---

†Strictly speaking, this construction is not permitted by the concrete syntax, for the result-type is neither a mode-constant nor an identifier (c.f. §5.12.1). The above usage is intended only to be notationally suggestive.

mval ~

PROC(p: ptr_any) mode;

mval(p) is the mode of the object to which p points. Note that the type-conversion rules for pointers are such that mval(p) is well-defined for any p whose mode is of class ptr. One special case of mval bears mention: if p = NIL then mval(p) = NONE.

5.15.4 Code Operations

xct ~

PROC(c: machine_code, name_pdl: pdl, pdl_index: int) ptr_any;

This executes the block of machine code c in the environment defined by name-pdl and pdl-index, producing a value $\mathcal{O}$. The result of xct is a pointer to $\mathcal{O}$.

5.15.5 Input/Output

The I/O routines assume that there exist two files – one for input and one for output.

read_form ~

PROC( ) form;

Reads one form. (This uses the parser discussed in section 4.3.1 and performs transformation to internal representation; c.f. section 5.1.3.)

write_char ~ PROC(x: char) none;

write_int ~ PROC(x: int) none;

write_bool ~ PROC(x: bool) none;

Each of these three procedures writes an object of fixed type onto the output file. The output is such that read-form will read in the object correctly.

5.15.6 Mode Operations

build_row_assigner ~

PROC(m: mode, l: int) proc_var;

Constructs and returns a procedure which performs assignment for objects of type $\mathfrak{M}$ defined by m and l as follows. If $l \geq 0$ then $\mathfrak{M}$ is ROW(l, m); otherwise $\mathfrak{M}$ is length unresolved ROW(m). The constructed procedure is equivalent in its action to

```
PROC(left: 𝔐 BYREF, right: 𝔐) none;
FOR i ← 1, ..., min(length(left), length(right)) DO
    left[i] ← right[i];
ENDP
```

build_struct_assigner ~

PROC(d: struct_def) proc_var;

Constructs and returns a procedure which performs assignment for objects of type $\mathfrak{M}$, defined by d in the following sense: if $\mathcal{O}$ has mode $\mathfrak{M}$, the i^{th} component of $\mathcal{O}$ has mode d[i].type (c.f. §5.9.4 for a definition of struct-def).

The constructed procedure is equivalent to

```
PROC(left: 𝔐 BYREF, right: 𝔐) none;
FOR i ← 1, ..., length(d) DO left[i] ← right[i];
ENDP
```

build_row_selector ~

PROC(m: mode, l: int) proc_var;

Constructs and returns a procedure which performs selection on objects of type $\mathfrak{M}$ defined by m and l as follows. If $l \geq 0$ then $\mathfrak{M}$ is ROW(l, m); otherwise, $\mathfrak{M}$ is length unresolved ROW(m). The constructed procedure has the heading

PROC(x: $\mathfrak{M}$ BYREF, i: int) m;

It returns the i^{th} component of x provided that $1 \leq i \leq$ length(x); otherwise, it calls error ("selection-error").

build_struct_selector ~

PROC(d: struct_def) proc_var;

Constructs and returns a procedure which performs selection on objects of type $\mathfrak{M}$, defined by d in the following sense: if $\mathcal{O}$ has mode $\mathfrak{M}$, the i^{th} component of $\mathcal{O}$ has mode d[i].type. The constructed procedure has the effective heading

PROC(x: $\mathfrak{M}$ BYREF, i: int) d[i].type;

It returns the i^{th} component of x provided that $1 \leq i \leq$ length(d); otherwise, it calls error ("selection-error").

build_ptr_assigner ~

PROC(m: mode) proc_var;

Constructs and returns a procedure $\mathscr{P}$ which performs assignment to objects of mode $\mathfrak{M} \equiv$ PTR(m). $\mathscr{P}$ has the heading

PROC(left: $\mathfrak{M}$ BYREF, right: $\mathfrak{M}$) none;

Note that both arguments to $\mathscr{P}$ must have mode $\mathfrak{M}$.

build_united_pointer_assigner ~

PROC(ms: modeρ) proc_var;

Constructs and returns a procedure $\mathscr{P}$ which performs assignment to objects of mode $\mathfrak{M} \equiv$ PTR(ms[1], ms[2], ..., ms[length(ms)]). If we define $\mathfrak{M}' \equiv$ RANY(ms[1], ms[2], ..., ms[length(ms)]) then $\mathscr{P}$ has the heading

PROC(left: $\mathfrak{M}$ BYREF, right: $\mathfrak{M}'$) none;

Note that $\mathscr{P}$ performs two conceptually distinct actions: (1) copying the pointer right into the pointer left, (2) arranging that dynamic type of left is properly adjusted such that mval(left) = mval(right).

5.15.7 Storage Management

allocate ~

PROC(m: mode, d: intρ) PTR(m);

Obtains from the heap a block of storage large enough to hold an object with mode m and dope vector d. All length fields which occur in the block are given values in accord with d and the block is initialized to the default value of m by a call on initialize (c.f. §5.14). The result of allocate is a pointer to the block, i.e., the allocated object. If a satisfactory block cannot be obtained from the heap, reclaim is called. If afterward a block still cannot be obtained, error ("allocate-error") is called.

reclaim ~

PROC( ) none;

Causes a garbage collection which reclaims all previously allocated blocks that are no longer referenced (c.f. §4.3.2).

length2 ~

PROC(p: ptr_any, i: int) int;

Obtains the i^{th} element of the dope vector which describes the object to which p points. If there is no i^{th} element of that dope vector, length2 is undefined.

5.15.8 Stack Operations

get_stack_space ~

PROC(s: stack_ptr, m: mode, d: intρ) ptr_any;

Identical to allocate(m, d) except that

(1) the storage is obtained from the STACK to which s points, rather than from the heap,

(2) the result is a ptr-any which references the allocated object rather than a PTR(m),

(3) if there is insufficient space in the STACK to satisfy the request, error ("stack-space-error") is called.

free_last ~

PROC(s: stack_ptr) bool;

Frees the last block obtained from the STACK to which s points and returns TRUE, provided that the STACK is not empty. If the STACK is empty, free-last returns FALSE.

last_in ~

PROC(p:ptr_any, s: stack_ptr) bool;

Returns TRUE iff the object pointed to by p is (or is part of) the most recently allocated block in the STACK pointed to by s.

5.15.9 Miscellaneous

pack ~

PROC(s:string) symbol;

Returns a pointer to the symbol table element (c.f. §5.5.3) for the identifier whose printed representation is the string s. If no such element exists, one is created.

install_initial_environment ~

PROC(name_pdl: pdl, index: int, value_stack: stack_ptr) none;

Installs the initial environment in which evaluation of EL1 programs takes place by putting on the name-pdl and value-stack initialized variables for all procedures which may be used in the language without definition; i.e., eval and all procedures in §5.15, 16.

error2 ~

PROC( ) ptr_any;

The inner routine for processing errors. Its action depends on the implementation.

5.16 BUILTIN PROCEDURES

In the interest of convenience and efficiency, a number of procedures which are not logically primitive are predefined. That is, they exist in the initial environment and may be used without definition by an EL1 program. (They can, however, be redefined by the program.)

The definitions given here should be taken as specifying the result of a builtin procedure; for reasons of efficiency, these results may be actually obtained more directly than is indicated by the definition. However, the actual procedure is strongly equivalent to the defined procedure: it produces identical results and side effects.

5.16.1 Logical Operations

not ←

PROC(p: bool) bool;

p ⇒ FALSE; ELSE TRUE;

ENDP;

```
∧ ←
PROC(p:form UNEVALED, q:form UNEVALED) bool;
eval_to_type(p, bool) ⇒ eval_to_type(q, bool);
ELSE FALSE;
ENDP;

∨ ←
PROC(p:form UNEVALED, q:form UNEVALED) bool;
eval_to_type(p, bool) ⇒ TRUE;
ELSE eval_to_type(q, bool);
ENDP;
```

5.16.2 Equality Tests

The two tests = and ≠ require a mode definition

```
simple_type ⇐ RANY(int, bool, char, ptr_any);
```

Using this we define

```
= ←
PROC(x:simple_type, y:simple_type) bool;
typ(x) ≠ type(y) ⇒ FALSE;
typ(x) = int ⇒ equal_int(x,y);
typ(x) = bool ⇒ equal_bool(x,y);
typ(x) = char ⇒ equal_char(x,y);
typ(x) = ptr_any ⇒ equal_ptr(x,y);
ENDP;

≠ ←
PROC(x:simple_type, y:simple_type) bool;
not(x = y)   ENDP;
```

5.16.3 Arithmetic Operations

```
≥ ←
PROC(x: int, y: int) bool;
x > y ⇒ TRUE;
ELSE equal_int(x, y);
ENDP;
```

The binary operators ≤ and < are defined analogously.

```
sign ←
PROC(x: int) int;
x > 0 ⇒ 1;
0 > x ⇒ -1;
ELSE 0;
ENDP;
```

5.16.4 Input/Output

```
read ←
PROC(m: mode) m;
NT read(m) returns an object of mode m. For example, 3+read(int) is legal;
eval_to_type(read_form( ), m);
ENDP;
```

```
write_symbol ←
PROC(s: symbol) none;
NT write_symbol(s) outputs the print_name of s as a sequence of
    characters;
FOR i ← 1, ..., length(val(s).print_name) DO
    write_char(val(s).print_name[i]);
ENDP;
```

5.16.5 Miscellaneous

```
typ ←
PROC(x: form UNEVALED) mode;
NT typ(x) returns the mode of x;
mval ∘ eval(x);
ENDP;

length ←
PROC(x: form UNEVALED) int;
NT length(x) is the number of components in x, provided that x is a row.
    If x is a pointer it is dereferenced and length of the result is taken.
    If x is neither a row nor dereferenceable to a row, then an error occurs;
DECL p: ptr_any;
p ← dereference ∘ eval(x);
mval(p).class ≠ "row" ⇒ error("length_error");
mval(p).d.length ≥ 0 ⇒ mval(p).d.length;
ELSE length2(p, 1);
ENDP;
```

5.17 INDEX TO SECTION 5

Given below are indices to the procedures and modes used in the formal specification of section 5. The citation gives the sub-section in which the formal definition occurs.

5.17.1 Procedures

5.17.2. Modes

Section 6. SYSTEM AND IMPLEMENTATION ISSUES

In this section we continue the topic begun in section 4: describing those aspects of the language and the system in which it is embedded which are not amenable to formal specification. Whereas section 4 dealt with fairly general topics, here we treat more specialized issues whose presentation requires concepts or definitions developed in the formal specification.

6.1 ASSIGNMENT FUNCTIONS, SELECTION FUNCTIONS, AND STORAGE FORMATS

Implementing the assignment and selection functions in an open-ended data type scheme requires interaction between (1) the construction functions, e.g., build-row-selector, and (2) the internal format of data storage blocks as established by allocate and the stack manager. To discuss the internal format, we begin with the role of the dope vector in establishing object sizes.

Let $\mathfrak{M}$ be a mode with n length unresolved dimensions; i.e., $\mathfrak{M}$.dope-length = n (c.f. §5.9.5). Hence, to completely specify an object $\mathcal{O}$ of mode $\mathfrak{M}$, n distinct lengths must be supplied. If $\mathcal{D}$ is a vector of i integers (i.e., typ($\mathcal{D}$) = INT and length($\mathcal{D}$) = n) then $\mathcal{D}$ and $\mathfrak{M}$ specify an object $\mathcal{O}$ as follows. In a top-to-bottom, left-to-right treewalk of the descriptor of $\mathfrak{M}$, let the i^{th} unresolved dimension be $\mathcal{D}$[i]. In the degenerate case where $\mathfrak{M}$ is length resolved, n = 0 and $\mathcal{D}$ has length 0.

The following convention is used in establishing the internal format of all data storage blocks: if $\mathcal{O}$ is an object with mode $\mathfrak{M}$ and dope vector $\mathcal{D}$, then associated with $\mathcal{O}$ is a set of n integers which specify dope information. In obtaining the block for $\mathcal{O}$ on either the heap or a stack, the system provides the additional storage this requires and sets up the dope

information in the appropriate positions.† Subsequently, the dope information cannot be changed but can be interrogated by the primitive function length2 (c.f. §5.15.7). Several points should be noted.

(1) If $\mathfrak{M}$ is fully length resolved (e.g., ROW(3,INT)) then no dope information is carried with $\mathcal{O}$, so that modes of maximal binding are handled efficiently.

(2) The set of n integers which is carried with $\mathcal{O}$ to specify dope information is <u>not</u> an intρ. If it were, a dope vector would be required to specify the length of the intρ; consistency leads to an infinite regress.

(3) In some cases, this scheme is quite redundant. Since each component of $\mathcal{O}$ will have its own dope information if its mode is length unresolved, identical information may appear in several places; e.g., consider R(R(CHAR)). However, the additional storage occupied will in general be a small price to pay for the generality and speed this scheme affords.

In the case of a <u>row</u> in which the mode of the components is length unresolved, one additional piece of information is stored with the object: the component size. This specifies the length, in machine-dependent storage units, of a single component and is calculated when the row is created. It is used by the selection function in calculating the location of the i^{th} component.

---

†Calculation of the amount of storage required and initialization of this storage may be carried out either (1) by a single routine which takes $\mathfrak{M}$ and $\mathcal{D}$ as arguments, or (2) by a set of routines, one for each mode, taking only the single argument $\mathcal{D}$. In the latter case, the routine is constructed when the mode is defined. In this case, the ddb is defined to have an additional component, "storage-fn", which holds the routine. Use of a storage function for each mode requires more space and is somewhat more difficult to implement than a single routine; however, the faster operation it provides will very likely make it worthwhile.

To complete description of the storage format, it is necessary to specify how the components of rows and structs are stored. Rows are simply stored sequentially; however, the layout of structs is somewhat more complicated. If all components (or all but the last) are length resolved, then they are stored sequentially; otherwise, each component which is not length resolved is replaced by an internal pointer which points to the location in the block (relative to the block origin) where the component is actually stored. Components which are so replaced are actually stored sequentially following all the others. For example, consider

```
m ← STRUCT(field1 : CHAR,
           field2 : R(complex),
           field3 : R(BOOL),
           field4 : INT,
           field5 : R(PTR_ANY))
```

An object $\mathcal{O}$ of mode m consists of

(1) a dope vector for $\mathcal{O}$ consisting of 3 integers,

(2) a CHAR, i.e., "field1",

(3) an internal pointer to (8),

(4) an internal pointer to (10),

(5) an INT, i.e., "field4",

(6) a dope vector for "field5",

(7) a R(PTR_ANY), i.e., "field5",

(8) a dope vector for "field2",

(9) a R(complex), i.e., "field2",

(10) a dope vector for "field3",

(11) a R(BOOL), i.e., "field3".

The internal pointers make it possible to obtain the i^{th} component of a struct without calculating the size of all $j = 1, 2, \ldots, i-1$ components.

Given the storage format, techniques for calculating the storage mapping function are well-known. An algorithm is given in [Hoff62] and an improved version in [Deu66] for the special case that all primitive data types require an equal number of storage units and layout is strictly sequential. Adaptation of this algorithm to differing size primitive types and to use of internal pointers is straightforward.

The procedures generated by the constructors build-row-assigner, build-row-selector, build-struct-assigner, and build-struct-selector are bound as tightly as the mode definition permits. Struct selection is always immediate, either because the displacement of the i^{th} component is constant or because an internal pointer is used. Row selection of the i^{th} component is by a displacement $k \cdot i$ where k is either a constant for the mode or is stored with the object. In general, whenever a length is known, that knowledge is reflected in the storage functions. For example, if a mode is fully length resolved, the assignment function is a routine which simply copies the appropriate number of storage units; this is a very short loop or possibly a single MOVE instruction on some machines. Similarly, the check of subscript range in selection functions uses a constant whenever the number of immediate components is fixed for that mode.

6.2 EVALUATOR RECURSION

In section 5, the semantics of EL1 were specified by an evaluator. To use the evaluator as an interpreter for running programs on a computing machine, it must be coded into some language which already runs on that

machine. Typically, this will be assembly code. The recoding presents only one real problem: all EL1 procedures are inherently capable of recursion and recursion is used extensively in the evaluator. Some programming discipline is needed to obtain assembly language routines which can operate recursively.

The essential requirement is that nothing in the program region is modified by the program; all storage which would normally lie in the program region is placed on a stack created in a separate data region. It is convenient, in fact, to use two stacks. One, called the control stack, contains return addresses for all subroutine calls which have been made but for which no return has yet occurred, along with some other control information. The other stack contains internal variables; i.e., working storage and the saved values of registers at higher levels. To decrease the number of stacks in the system, it is useful to make this stack and the name-pdl (c.f. §5.3.5) the same, with entries for internal variables and program variables intermixed. Since the evaluator requires no names for its internal variables, the "name" component for these entries can be empty or filled in with special system names which do not clash with ordinary variable names.

Note that these requirements are almost identical to those for re-entrant code which can be used by multiple processes ("tasks" in IBM 360 parlance). One copy of the interpreter shared among several EL1 processes amounts to a nice saving of core. Hence, whenever the operating system does not make this impossible, the interpreter will be coded and interfaced to allow re-entrant use.

One related point should be mentioned here. The formal definition specifies that the value-stack, return-slot, and aux-return-slot (c.f. §5.3.5)

are STACKs in the sense of section 3.16.3. This is to guarantee, for the purpose of formal definition, that these stacks behave correctly, i.e., that the evaluator never uses a pointer to an object which has been freed. In the implementation, the STACKs will be replaced by ordinary LIFO stacks. Guaranteeing that stack discipline is not violated is a burden placed on the implementor.

6.3 STACK OPERATIONS

Although STACKs are not to be used in implementing the value-stack, the mode is in the language for use by the programmer. Implementing the STACK operations is, for the most part, straightforward. Performing sub-allocations within a STACK, freeing sub-allocations, and testing whether a pointer references the "top" sub-allocation present no great problems. The only real issue is guaranteeing that all PTR-ANYs which reference freed sub-allocations have the value NIL.

A number of possible implementation techniques present themselves. The best of these appears to be connecting all pointers to a sub-allocation into a doubly linked ring which passes through a control block associated with that sub-allocation. When the sub-allocation is freed, it is only necessary to walk around the ring setting each pointer to the value NIL. The ring is established and maintained as follows; whenever as assignment of PTR-ANYs such as

p1 ← p2

is made, p1 is removed from its ring and added to p2's ring. Typically, only one pointer assignment will be made per sub-allocation: the one which occurs immediately after the call on get-stack-space. Hence, the overhead due to ring manipulation should be small.

Section 7. CRITICAL DISCUSSION

In this section we carry out a critical analysis of EL1 and its formal definition. This analysis has several facets. We begin by describing and justifying certain design decisions and examine how these relate to each other and to the design goals. We draw attention to certain language features, the reasons for their presence, and their consequences. We compare EL1 to other programming languages and analyze the causes of their differences. Finally, we assess the formal definition of section 5, in particular, comparing it to the methods of semantic specification discussed in section 2.1.

In analyzing the base of an extensible language, there are two targets which can be addressed:

(1) the base language as it stands, treating it as if it were a fixed, conventional programming language,

(2) the family of languages which can be produced from this base by extension.

One of the chief virtues of an extensible language system is that (1) is unimportant compared to (2), the point of view being: if you don't like it, change it. Hence, our analysis ignores issues such as syntax[†] in favor of basic issues underlying the structure of the language. Our concern is with substance, not style.

[†]This is not to say we believe syntax is unimportant. Quite the contrary, we have given considerable attention to the syntax of EL1 and are particularly pleased with its present form. However, we recognize that choice of notation is subject to personal idiosyncrasy; what we find pleasing may repel another programmer.

A word concerning the division of this section into subsections is perhaps in order. While we have placed analysis and comparison in separate subsections, there is considerable overlap in their potential range. The analysis of language features can be aided by examination of their counterparts elsewhere, while the pertinent comparison of languages is carried out not by parallel listings of similar concrete forms but rather by critical analysis of underlying structures and their interrelation. Recognizing this, the division of topics between the two subsections should be taken as somewhat arbitrary. We have placed a topic in the second when its analysis depends primarily on comparison and in the first otherwise; however, this is only a matter of emphasis.

7.1 ANALYSIS AND JUSTIFICATION OF LANGUAGE FEATURES

7.1.1 Basic Objects and Storage Blocks

In section 3.15, we noted that as one goes deeper into the study of data types, the model of storage behavior employed becomes increasingly more important. In that section we argued that recursive modes should be excluded from EL1, on the grounds that they were in various ways non-primitive. Here, we generalize these considerations.

The principal axiom which shapes the data type facility in EL1 is that each basic object resides in a contiguous block of storage which is effectively† fixed during the existence of that object. By basic object, we mean any object whose mode can be directly defined in the language using

---

†No prohibition is here intended against an implementation moving objects, e.g., to compactify storage. However, the language can in no way depend on such movement.

the mode definition facility of section 5.9. This is, of course, a very restrictive view of permissible basic objects, but it corresponds well to the primitive operations and architecture of current computing machines. For example, we observe that the notion of row is in one sense subsumed by linked list; however, accessing the i^{th} element of a row is a primitive operation, while it requires time i in a linked list. The identification of objects with storage blocks makes implementation in a stack immediate and simplifies heap implementation, for an n-word block is an elementary, easily manipulated entity. We contend that this is the only admissible representation for basic objects. In so doing, we insure that the data types on which extensions are built have behavior which is well-matched to the available hardware. The importance of an impedance match here should be obvious.

In consequence, the mode definition facility of EL1 is restrictive. Not only do we rule out recursive modes, but also arrays with flexible bounds. The latter are admissible in various forms in several languages, notably: GPL [Gar68], Basel [Jorr69], and Algol 68 [vanW69]. There are, of course several strategies for implementing such arrays, with various trade-offs between them depending on the expected traffic. GPL and Basel each use a different strategy, carry around the mechanism to implement the chosen strategy, and provide no control over this to the programmer. However, this is precisely the sort of inflexibility and overhead one sought to escape in an extensible language. We argue that the proper approach is to obtain extendible arrays by extension – several extensions for several different traffic patterns.

Other classes of non-basic objects will similarly be obtained by extension. That is, taking basic objects as building blocks, one constructs

distributed objects and higher-level objects by pointer-linkage, remapping functions, and similar formation rules. To describe such objects, we require higher-level modes which cannot be defined directly by the existing mode definition facility, but which can be constructed from basic modes with a small amount of additional mechanism. We present this mechanism as a metaphrase extension in section 8.4. Here, we wish to emphasize that restricting the basic objects does not restrict the semantic space of the language; it only sharpens the base component.

7.1.2 Global Storage Model

Just as the contiguous-fixed-block requirement shapes the treatment of modes and the local issues in storage, certain global aspects of the storage model shape other portions of the language. As discussed in section 4.3.2, EL1 uses both a stack and a heap for its storage. Stack storage is strictly governed by procedure entry and exit; heap storage is created by calls on allocate and reclaimed by garbage collection.

Some consequences are obvious. Provision for dynamic storage allocation in the heap allows list processing and opens the language to numerous important applications which require list processing: artificial intelligence, symbol manipulation, and graphics, to name a few. Linking the stack storage to procedure calls admits recursion but makes somewhat difficult any call structure other than nested procedure calls.†

---

†Note, however, that other call structures are not excluded. Many can be obtained by extensions. To obtain co-routines, for example, it is necessary to provide (1) storage which is not destroyed when the co-routine is exited, (2) record of where a co-routine is exited, (3) a call structure which recognizes a co-routine call and handles it specially. The heap provides storage for (1) and (2) so that the only real issue is (3) and this is independent of the storage model.

However, other consequences are more subtle. For example, since a garbage collector is used, it is necessary to check subscript range on access to rows. Indeed, were an assignment made out of range, it would clobber some random word in the address space; if the word, or part of the word, were in use as a pointer and were traced, the garbage collector would be hopelessly fouled. While out-of-range register clobberings may occur in any language, the existence of a garbage collector makes it a global and hence unacceptably dangerous error. The use of garbage collection is also partly responsible for the policy of initializing all objects and for inclusion of the function length. While each can be justified on other grounds, garbage collection would alone suggest their appearance. In all blocks allocated to hold variables, pointers must be initialized to NIL, otherwise random trash may be erroneously traced. The obvious generalization is to initialize all objects. Objects of length unresolved mode must carry a length field for use by the garbage collector; since the length field exists, we give the programmer (fetch only) access to it.

A deeper consequence stems from the protection produced by garbage collection. Since storage cannot be explicitly returned to the heap, it is impossible to return an object having forgotten that a pointer p still references it. This is easy to do in a system which allows explicit return. (The return alone causes no trouble, but if the space is later reused, then fetches through p produce trash and assignments clobber random words.) A garbage collection system prevents such errors occurring when the referenced block is in the heap. However, it may still be possible to produce a similar error if there is a pointer into some area of the stack which is freed (on procedure exit) and later reused (for another procedure activation). Precisely this sort of situation and error can occur in Algol 68.

If such a misuse of pointers occurs, the official Report specifies that the program is undefined, but the error may not be detected in all implementations. EL1, on the other hand, provides complete protection against storage violations by prohibiting pointers into the stack. That is, the language is so designed that there is no way of getting a pointer to reference a stack location (cf. §3.9.4). While this excludes from EL1 a number of constructions found in Algol 68, many of these are not really useful. Most of the useful ones are obtained in EL1 through other mechanisms which permit protection against their misuse (e.g., cf. § 7.2.2).

To conclude this subsection, it may be useful to question, rhetorically, our choice to use a garbage collector. Three alternatives present themselves: (1) to allow explicit freeing but also garbage collect, (2) to require explicit freeing as the only way of recovering storage, (3) to use some other automatic reclamation technique. Proposal (1) has been tried with disappointing results [Bobr69] . When an explicit free command was added to Lisp 1.5, it was found to be frequently used illegally. That is, blocks were inadvertently freed while pointers to them remained active; the resulting pointers into the free-list caused havoc. Proposal (2) is more consistent; placing all burden of reclamation on the programmer clearly fixes the responsibility. However, the burden may well prove too heavy. Our experience has been that algorithms which make extensive use of heap storage are sufficiently complex that the suppression of bookkeeping provided by garbage collection is well worth the price. Turning to (3),

the difficulty is that alternative methods of automatic reclamation are not altogether satisfactory. The best of these, reference counts (e.g., cf. [Coll66]), fails to return block sets forming a self-referencing net. Since such nets may be expected to occur frequently, the resulting loss of storage will be unacceptable. Further, reference count systems are particularly poor in paged environments where the act of fetching the blocks referenced in order to change their use counts may involve additional references to secondary storage, at considerable cost in time.

7.1.3 Primitive Modes and Mode Definitions

Of the seven primitive modes in EL1 – INT, BOOL, CHAR, STACK, PTR-ANY, NONE, and NONEREF – all but the first three are unconventional. The motivation for STACK has been discussed in section 3.16.3; here, we treat PTR-ANY, NONE, and NONEREF.

A PTR-ANY is a general handle on the heap. As it may reference any allocatable object, it allows the writing of procedures which can manipulate literally any sort of data object. As such, PTR-ANY provides a weak, i.e., one pointer level off, substitute for the data type any which has been proposed in some extensible languages.

Implementing PTR-ANY is fairly straightforward. One technique is to allocate for each PTR-ANY enough storage to hold two addresses – one for the location of the object referenced and one for the location of the object's ddb; the second address carries the type information. Alternatively, it is possible to divide the heap into

quantum zones, each containing a unique mode of object (cf. § 4.3.2); in such a system, each PTR-ANY requires storage to hold only one address, for the type information is carried implicitly in the address of the object.†

However, code generation by a compiler for PTR-ANY's presents a problem. Certain special cases such as the top-level occurrence of

mval(p) = amode ⇒ BEGIN . . . END

allow determination of mode in local regions (here, in the compound-form) and hence make possible reasonable code generation. However, in general, the compiled code will be heavily interspersed with calls on the interpreter.

---

† This requires some refinement when an object contains components (whose mode differs from that of the object) and it is desired to point to a component; the mode of the component is not in agreement with the region in which it is located. In such cases, it is necessary to have the PTR-ANY address a special link pointer which contains two addresses — one for the component and the other for the component's ddb. That is, this scheme attempts to use a single pointer with quantum zone data typing whenever possible and uses a modification of the other scheme as a fall-back position when this is not possible. This technique will be more efficient if and only if the fall-back position is taken only infrequently.

The modes NONE and NONEREF verge on triviality, for they each have a single value: NOTHING and NIL, respectively. While their existence is slightly displeasing, as they raise the number of primitive modes from a reasonable 5 to a suspiciously large 7, they appear to be unavoidable. Since every procedure has a result, there must be some way to indicate that a result is of no interest and should be ignored, hence the mode NONE to use as a return-type declarer. A mode is barren without values, hence the value NOTHING.

The mode NONEREF, on the other hand, arises from its value, NIL. We need NIL as a dummy value for pointers. Otherwise,† each object of mode PTR($\mathfrak{M}$) requires some $\mathfrak{M}$ to point to; this may be undesirable when $\mathfrak{M}$'s are large. The value NIL requires a mode, hence NONEREF. Note that EL1 does not identify the notions of NOTHING and NIL. Hence, it is possible to distinguish between (1) a procedure returning NIL, a value which may be validly assigned to a pointer, and (2) a procedure which returns NOTHING. Blurring this distinction would, we believe, merely weaken the language.

Several of the data type definitions used in the evaluator of section 5 perhaps require discussion, for they use a common device that may be of general utility: representing a notion by the data type PTR(notion). For example, (1) a symbol is defined as a ptr to a symbol table entry, (2) a mode is defined as a ptr to a ddb which contains the mode definition,

---

†A third possibility is to have a distinctive nil value for each mode, i.e., a PTR($\mathfrak{M}$) either points to an $\mathfrak{M}$ or the nil-$\mathfrak{M}$. However, this appears to be more complex than a single NIL and offers no compensating advantages.

(3) a proc-var is a ptr to an explicit procedure which is the defining block. We use the semantically loaded term, not to name the mode of the objects which directly represent the term but rather to name the mode pointer-to-such-objects.

This is done partly for reasons of efficiency. By using symbols instead of character strings to represent the notion of symbolhood, we allow names of unlimited length, speed up equality tests, simplify storage management, and usually save on storage. Similarly, a pointer to a ddb is generally as satisfactory as the ddb itself, but uses far less space.

This pointer transformation is also due in part to the fixed-block axiom discussed in section 7.1.1. The axiom fares poorly if we attempt to apply it to variables of certain intuitively reasonable data types, most notably "procedure". That is, if p is declared a procedure-valued variable, we would like it to hold any procedure. Since the defining blocks come in various sizes, the correspondence of procedure-valued variable and defining block fails. As a fallback position, we form the correspondence between procedure-valued variable and a ptr to the defining block. It should also be noted that, quite aside from assigning semantically loaded words, the use of PTR(notion) is an example of one of the most widely applicable sorts of distributed objects (cf. §7.1.1).

One additional point should be raised in conjunction with the representation of "procedure" by data types. In EL1, procedure-valued variables are only weakly typed; i.e., having declared the variable p to be a proc-var, we can assign to it any procedure. In contrast, declarations of procedure-valued variables in Algol 68 restrict the sorts of procedures

which can be so assigned. For example, consider†

DECL p : PROC(INT, complex) CHAR ;

The variable p can be assigned only those procedures which accept two arguments, an INT and a complex, and deliver a CHAR. We say that in Algol 68, procedure-valued variables are strongly typed.

While this difference of Algol 68 and EL1 may appear striking, it arises in a natural way from the evaluator models assumed by the languages: a compiler and interpreter, respectively. To perform type checking when compiling code for calls on p, one wishes to know how many arguments it is supposed to take and of what mode. Hence, Algol 68 constrains the variability of p. If, however, interpretation is performed, there is no need to impose such restrictions. We point out this difference in the two languages primarily to minimize its significance. Ultimately, a compiler will be written for EL1. To allow reasonable compilation, we will impose certain requirements on programs to be compiled. One such restriction will be strong typing of proc-vars.

7.1.4 L-Values

EL1 is unique among algorithmic languages in that all values are L-values; i.e., the locative condition holds. In section 3.13, we discussed the locative condition and the language features to which it gives rise. Here, our concern is with analysis of the concept.

First, it should be recognized that the locative condition imposes constraints on the implementation. In programming languages not using

---

†To make this readable by those not familiar with Algol 68, we use the notation of EL1. We emphasize, however, that this illustrates the convention of Algol 68, not EL1.

the locative condition, the implementation is free to represent most values as either (1) the value stored in a register, say an accumulator, or (2) as a pointer to the value which is stored in memory. For example,

$$\underline{\text{if}}\ \text{b}\ \underline{\text{then}}\ \text{x}\ \underline{\text{else}}\ \text{y}$$

may either (1) leave in a full-word accumulator the value of x (or the value of y) or (2) leave in an index-register a pointer to the address where x (or y) is stored. The implementor is free to choose between (1) and (2), depending on the size of x and y, storage operations on his machine, and similar considerations. The locative condition, however, permits

$$(\underline{\text{if}}\ \text{b}\ \underline{\text{then}}\ \text{x}\ \underline{\text{else}}\ \text{y}) \leftarrow \text{z}$$

and hence prohibits use of (1), (except perhaps in compiled code where inspection of the surrounding program insures that (1) is "safe").

In consequence, the implementation space is constricted and certain common techniques cannot be applied. There may be special difficulty where an object is smaller than a machine word and does not begin on a word boundary. Depending on the degree of fineness in the machine's address structure and the degree of fineness required by the position of the object, an address may have to be supplemented by additional bits specifying interaddress precision. On the other hand, the restriction is not as onerous as it may first appear. For almost all machines, pointer representation is the only way of handling objects larger than one or two machine words. In a language such as EL1, where composite objects are common and can be transacted with directly, pointer representation will be the natural implementation technique for most situations.

In general, the advantages of the locative condition far outweigh the difficulties it causes. It makes possible efficient and uniform call by

reference, for given a pointer to the argument, the name and argument can be simply tied via an indirect-cell. Also, it simplifies the interaction of selection and assignment. Consider, for example, the two forms

```
x.a ← y
x.a + y
```

Note that the first form requires an L-value and hence the implementation must possess a selection function which yields an L-value. If we decide to use a nonlocative representation for x.a in the second case, then the implementation is forced to carry two distinct selection functions. Further, the evaluator is complicated by being required to pass along information as to whether a form is being evaluated as an L-value or an R-value.

In the semantic formalism of section 5, the locative condition is manifest as follows: an object $\mathcal{O}$ in the program being interpreted is handled by a PTR-ANY $\mathcal{P}$ of the evaluator. That is, the pointer assumed by the locative condition appears explicitly. Hence, the evaluator can change the contents of $\mathcal{O}$ or establish the binding pattern required by BYREF bindings with no difficulty.

The effect of the locative condition is very closely related to a convention adopted in Algol 68. Consider, for example, the EL1 declaration

```
DECL i : INT ;
```

or the equivalent Algol 60 declaration

```
int i;
```

To obtain the same effect in strict (cf. §2.2.2) Algol 68, one writes

```
ref int i = loc int;
```

This may be interpreted as follows. The external object i, which is an

identifier, corresponds to an internal object $\hat{i}$ of mode ref int; $\hat{i}$ contains a pointer to an object of mode int which has been allocated on the stack for locals. An Algol 68 assignment

```
i := 3
```

is then interpreted as

(1) take the contents of $\hat{i}$, this being the address of a position on the stack which has been reserved to hold ints,
(2) take the value 3,
(3) store 3 into the stack position obtained in (1).

That is, the assignment operator here takes the address of an int as its left operand and an int as its right operand. Algol 68 does not use the locative condition. Instead, it uses the data type ref $\mathfrak{M}$ where EL1 would use $\mathfrak{M}$. The two languages achieve similar effects by different devices. EL1's method seems to permit simpler and more consistent languages, for it avoids the necessity of using a ref $\mathfrak{M}$ where intuition, inspired by the tradition of Fortran and Algol 60, would suggest an $\mathfrak{M}$.

7.1.5 Names and Values

As it currently stands, EL1 adopts the rule that in any given scope an identifier can be associated with at most one value. This is the convention adopted by Algol 60, Algol 68, and most other programming languages. However, in some languages (e.g., BBN Lisp 1.85 [Bobr68] and THE BRAIN [Win69]) it has been found convenient to allow an identifier to be associated with two or more values of different types. Context determines which value is intended. Typically, a fixed number of distinct values of prescribed types is permitted, with one value slot being devoted to procedures.

It is possible to go one step farther and allow different values to have different scope rules. For example, Bobrow [Bobr69] argues that humans tend to treat procedures differently from other sorts of values.

Procedures, he argues, are generally thought of as global entities, defined at the outermost level, and having scope throughout the entire program. The realization of this in BBN Lisp is that the procedure corresponding to a given identifier is found by checking the top-level function-cell associated with the symbol table entry for that identifier. A second-order scope rule provides that if the function-cell is found to be empty, then the "normal" value which has block structure scope rule is considered. For example, consider activation of the following procedure†

```
PROC(p: proc_var, x: INT) BOOL; p(x) ENDP;
```

In the form "p(x)", p will be taken to denote the top-level function-cell value of p if such a value exists; otherwise, p will be taken to denote the first argument to the procedure. That is, the procedure named "p" which is applied to x may or may not be the first argument. However, since context information is not passed down to subevaluations in BBN Lisp, the seemingly equivalent form

```
PROC(p: proc_var, x: INT) BOOL; [[TRUE ⇒ p; p]](x) ENDP;
```

is not equivalent; in it, the procedure p applied to x is always the first argument.

The merit of any given convention can be debated at length with a definite conclusion doubtful. (Indeed, despite the anomalies of the above examples, in our experience with the BBN Lisp convention we have found it quite satisfactory.) Suffice it to say that various programmers have diverse predilections, and it may well be that various applications suggest

---

†As usual, we use EL1 notation to convey a notion from a different language – here, BBN Lisp.

different conventions. In the face of such potential variation, EL1 uses the simplest rule available to associate names and values. We take the position that should this prove unsatisfactory, metaphrase extension is the appropriate remedy. In section 8.3, we discuss one such extension.

7.1.6 Dynamic Treatment of Modes

The feature of EL1 most at variance with tradition is its dynamic treatment of modes. Languages, extensible or otherwise, having data types are typically compiler-based and perform most transactions with modes at compile-time. Indeed, by run-time mode information has usually been completely discarded. Disregarding for the moment those activities of the compiler which make a computing machine act as an executor of source language statements, there remain two distinct phases in evaluating a program: mode-processing at compile-time and statement-processing at run-time, with quite different evaluation rules. In simple languages, the mode processing is relatively trivial and can often be ignored. However, in larger languages (such as PL/I) or in extensible languages, the data type declaration rules are complex and verge on becoming a (non-executable) language in their own right. The mode evaluator becomes correspondingly complex; e.g., consider the treatment of mode recursion and forward reference discussed in section 3.15.

Our concern in the design of EL1 was with maintaining simplicity and parsimony. The use of a special and rather complicated evaluator exclusively for modes held little appeal. The semantic specification would be larger, communication of the language more difficult, and implementations would be forced to carry more code. Further, a special disjoint mode language would be an inevitable target for accretions. That is, with any

sub-language there is an irresistible and often valid pressure to add to it features found in the main language, thereby increasing the expense of carrying the sub-language. Hence, we decided to use a single evaluator and force everything into its Procrustean bed. By and large, the experiment was successful: little was amputated, and the stretch was beneficial.

We have already discussed the undesirable consequences in section 3.15; here, we concentrate on the advantages. By allowing the standard evaluator to be used for modes, we permit the application of powerful formation rules to the definition of data types, notably conditionals, iteration, and procedure recursion. We illustrate these in turn. Consider, for example

```
special_string ← [[ max_record ≤ k ⇒ ROW(max_record, CHAR);
                    ROW(CHAR) ]]
```

The mode special-string is either a row of max-record CHARs or, if this requires reserving too much storage for each special-string, it is a length unresolved string. In general, conditional computation of mode definitions allows forms which compute optimal definitions perhaps based on information known only after the time the definitions are written.

Iteration allows procedures such as

```
complete_binary_tree ←
PROC(depth: INT, m: mode) mode;
DECL temp : mode;
temp ← m;
FOR i ← 1, . . ., depth DO
    temp ← S(left: temp, right: temp, value: m);
temp ENDP;
```

The result of the procedure is a mode: the complete binary tree of depth levels where each node has a "value" cell of mode m and two sons called "left" and "right". This one procedure defines and describes a family of modes and serves as a realization in the language of the intuitive notion "complete binary tree".

The descriptive power alone is sufficient motivation for allowing strong formation rules. For example, turning to procedure resursion, we have

```
array ←
PROC(m: mode, order: INT) mode;
order = 0 ⇒ m;
ELSE ROW(array(m, order-1));
ENDP;
```

which delivers the mode: array of dimension order m's. For example,

```
string ← array(CHAR, 1);
bool_matrix ← array(BOOL, 2);
third_order_tensor ← array(INT, 3);
```

It is important to note that array describes only one method of representing multidimensional objects. In implementing GPL, Garwick used a different representation [Gar68b] which may be described as

```
garwick_array ←
PROC(m: mode, order: INT) mode;
order = 0 ⇒ m;
order = 1 ⇒ ROW(m);
ELSE ROW(PTR(garwick_array(m, order-1)));
ENDP;
```

Every garwick-array of order >1 is a row of pointers; hence, components can be shared among two or more garwick-arrays whenever the components are compatible. The point we wish to make is that the use of such procedure definitions allows precise, concise specifications of mode families.

Sets of mode-valued procedure definitions which call each other conditionally and themselves recursively behave effectively as one form of high-level mode definitions. With their use it is easy to define modes of considerable complexity, efficiency, and power. Hence, we recover the power apparently lost by restricting basic objects while achieving generality and flexibility in the process.

7.2 COMPARISON WITH OTHER LANGUAGES

7.2.1 Bound Variables

We use the term bound variable as an antonym of free variable. A bound variable is either a formal parameter or a declared variable; a free variable is any variable which is not bound.

Few aspects of programming languages rouse so much confusion, controversy and acrimony as does the handling of formal parameters. There are at least four† sorts of bindings found in contemporary languages:

(1) by value, e.g. the Algol 60 call by value,

(2) by reference, e.g. the standard method of binding in Fortran IV,

(3) by name, e.g. the Algol 60 call by name,

(4) unevaluated, e.g. the binding of FEXPRs in BBN Lisp 1.85.

---

†An additional, orthogonal axis can be introduced if we consider different ways of spreading the argument list, i.e., of pairing arguments with formals. This at least doubles the number of possibilities. For simplicity, we ignore this issue and assume a 1-1, left-to-right correspondence.

These are characterized as follows. Bindings by value and reference both evaluate the argument; the former creates a new object and assigns the value of the argument to be the initial value of that object; the latter simply establishes an association between the formal name and the value of the argument. Binding by name and unevaluated both involve no evaluation of the argument. They differ in that after binding by name, a use of the formal parameter in the procedure body causes evaluation of the argument, whereas after binding unevaluated a mention of the formal parameter has as value the unevaluated argument. That is, the effect of binding by name can usually† be achieved with an unevaluated binding if every instance of the formal name, say "x", in the procedure body is replaced by "eval(x)", where eval is that language function which maps a form into its value.

Several languages attempt to achieve unification of concepts by subsuming two or more of these binding methods under some other. We have just shown how to subsume (3) under (4). It is also well known that call by name can be achieved by a call by value in which the argument passed is a procedure equivalent to the actual parameter which would have been passed by name. Indeed, this is the technique used for implementing the most general cases of call by name in Algol 60. Further, binding by reference can be subsumed under binding by value if there is an operator which maps any object into a pointer to that object. For example, in Euler [Wir66] where the operator is denoted by "@", the form "f(@b)" achieves a call by reference even though the binding mechanism is by

---

†The possible exception to this occurs when x appears on the left-hand side of an assignment statement and the locative condition (cf. §7.1.4) does not hold.

value, for "@b" is a pointer to b, and this pointer is assigned as initial value of the formal parameter.

Algol 68 uses a variation on this last technique. As noted in section 7.1.4, the practice in Algol 68 is to use a variable of data type ref $\mathfrak{M}$ where other languages would use one of type $\mathfrak{M}$. Consider, for example, the form "f(b)". If b would have type $\mathfrak{M}$ in Euler, it will generally have type ref $\mathfrak{M}$ in Algol 68. Hence, a pointer to the $\mathfrak{M}$ object is already available without need to apply the @ operator.† The binding for the Algol 68 "f(b)" may be carried out much like the binding for the Euler "f(@b)".

It might be assumed that EL1 would obey the dictates of simplicity and allow only one class of binding, e.g. by value. However, closer investigation reveals that the appeal to simplicity is deceptive. Having already imposed the locative condition, binding by reference is trivial to define and implement. Also, since arguments are initially unevaluated, binding unevaluated requires little more than omitting an application of eval. Hence, EL1 provides for value, reference, and unevaluated bindings. An inspection of bind-formals in section 5.13.4 will demonstrate that this adds little to the complexity of the evaluator. All the required mechanism is already present in the evaluator for other reasons so that its application here is merely an example of completeness. As a general principle, it would seem that whenever it is possible to cater to all tastes on a significant issue at little cost, the dictum of parsimony may be overridden.

---

†When the pointer is not needed, as in producing an effective binding by value, an automatic application of a val operation takes place. This is termed "dereferencing" and is a common occurrence in Algol 68.

The completeness argument is, however, a two-edged sword; it demonstrates a deficiency in the handling of declared variables in EL1. It should be recalled that declared variables are always initialized to mode-dependent default values. One might well argue for the utility of initial values obtained from a formal-actual correspondence analogous to that of formal parameters and their arguments. If the initial value is bound by value, an assignment statement following the set of declarations is an acceptable paraphrase. However, suppose we wish a binding by reference. For example,

```
DECL x : intρ EQU a[i];
```

where a is a two-dimensional array of ints. The intent is to establish that x names the same object as does a[i], the value of i being frozen at the moment of binding. This is analogous to a formal parameter x of mode intρ bound BYREF with argument a[i]. Utility and symmetry both argue that such bindings of declared variables be allowed. As EL1 currently stands, however, there is no EQU option permitted in a declaration. It should and will be added to the language in its next edition.

7.2.2 Free Variables

In section 3.17.2, we discussed the dynamic scope rule for free variables in EL1 while postponing its justification. Here, we deal with that issue. There appear to be only two alternatives to dynamic scoping:

(1) static scoping, as in Algol 60 or Algol 68,

(2) global scoping, as in APL, i.e. all free variables fall into a common, global pool.

Clearly, (2) is a subcase of dynamic scoping provided that no free variable names clash with bound variable names. Since the globals form a single

pool, this requirement can be readily satisfied by a simple scope-invariant renaming. Hence, the issue is dynamic vs. static scoping.

The following Algol 60 fragment should make the difference clear.

```
begin
    real d;
    real procedure f(x); real x; f := x + d;
    d := 1;
    y1 := f(10)
    begin
        real d;
        d := 2;
        y2 := f(10)
    end;
end
```

Using the dynamic scope rule, the first call on f identifies d with the d of the outer block and sets y1 = 10+1, while the second call on f identifies d with the d of the inner block and sets y2 = 10+2. Using the static scope rule, the d of the procedure is permanently identified with the d whose scope is the block in which the procedure is defined; hence, both calls add 10 to 1 (the value of the outer d). Note that Algol 60 uses the static rule. There are, however, several telling arguments in favor of dynamic scoping.

The first is based on simplicity. The dynamic scope rule gives precisely the correspondence which would be obtained if the procedure body were substituted in place of the procedure application. This so-called "copy rule" is the simplest, most consistent convention, for it preserves the scope rules of block structure in their sharpest form. Because there

is a uniform scope rule, the language is easier to learn and communicate.

The second point is that for most circumstances, dynamic scoping is more useful than static. It allows procedures written (and even compiled) separately to communicate via common names. More important, this communication remains valid during the execution of the program. That is, there is a single rule which associates meaning with a free variable x: the meaning of free-x-hood is always the most recent bound incarnation of x. If x is tied to some conceptual notion, references to "x" always obtain the local incarnation of that notion.

A third argument is based on the baneful effects of static scoping. To illustrate the problem, imagine adding to Algol 60 procedure-valued variables (procvars) and allowing assignment of procedures to such variables. Consider the following program (where lines are numbered for reference).

```
(1)  begin real a, d; procvar p1;
(2)      d := 10;
(3)      begin real b, d; procvar p2;
(4)          d := 20;
(5)          p2 := procedure (x); real x; x := x + d;
(6)          b := 1;
(7)          p2(b);
(8)          p1 := p2
(9)      end;
(10)     a := 1;
(11)     p2(a)
(12) end
```

The assignment on line (5) of a procedure to the procvar p2 is equivalent to

```
procedure p2(x); real x; x := x + d;
```

The static scope rule of Algol 60 would permanently identify d in the procedure with the d of the inner block. Hence, line (7) increases b by 20. The difficulty arises in line (8) which assigns the procedure to p1. This allows the procedure to be carried outside the scope of the inner d. Line (11) specifies that a is to be increased by the value of d in the inner block. However, that d, if implemented on the stack, has disappeared.

The problem does not arise in Algol 60 because procedures are not assignable values and hence have strictly static scope. As soon as we permit nonstatic scoping of procedures, the static scope rule for free variables encounters difficulties.

One solution, adopted by PAL [Evans69], is to completely abandon the stack. All storage is placed in the heap and storage blocks are reclaimed by garbage collection when no longer referenced. Hence, in the above example, the storage for the inner d exists so long as p1 or p2 exists, for the procedure which they reference references d. This sort of solution is not too unreasonable for PAL which is used only in an instructional capacity and hence for running small, short programs. However, its application in a language for general use would be seriously misplaced. Where a stack can be used, it is far more efficient than garbage collection. For most problems, this efficiency is far more important than the ability to obtain statically scoped free variables. To obtain the latter at the sacrifice of the former would be a poor trade.

Another solution,† adopted by Basel [Jorr69], is to associate the free variable with the value of the corresponding bound variable. For example, in the above illustration, the static scope rule of Algol 60 would associate "d" in the procedure with the location of the inner block's d; hence, assigning to "d" in the procedure would change the value of the inner block's d. The static scope rule of Basel, on the other hand, would associate "d" in the procedure with the value of the inner block's d at the time line (5) is reached, i.e., the real number 20. While this solves the problem, it changes the semantics. To obtain the sharing pattern of the Algol 60 original, it is necessary to declare the inner block's d to be a ref real and separately establish its value as a pointer to a real in the heap. Then, in line (5), the value of "d" in the procedure becomes the location of a real, this value being fixed. Since the heap real persists even after termination of the inner block, line (11) causes no trouble. However, note that this moves the real from the stack to the heap, with attendant overhead. That is, the PAL solution is adopted with two changes. (1) It is made explicit in the program. (2) It is used only where a bound variable object is to be shared with a free variable, so that the heap expense is not incurred for every variable. However, it is still an uneconomical, shotgun approach. Few cases of sharing between bound and free variables are coupled with a scope problem which really requires the heap, but all cases are forced to employ it.

---

†CPL has a similar feature, but only as an option. Procedures can be either "free" or "fixed". In the former case, free variables have dynamic scope. In the latter case, the value of the free variable is frozen at the time of procedure definition.

A third solution is adopted by Algol 68. It defines the scope of a procedure[†] to be the smallest scope of any free variable used within it. It then requires that in any assignment having the format

⟨procedure_valued variable⟩ := ⟨procedure⟩

the scope of the ⟨procedure⟩ must be at least as large as the scope of the ⟨procedure-value variable⟩. This guarantees that a ⟨procedure⟩ cannot be carried outside the scope of its free variables. The formal definition specifies that if a program attempts an assignment which violates this requirement, the further evaluation of that program is undefined. Clearly, the desired consequence in the case of such an illegal assignment is an error message, possibly in conjunction with a trap to an error recovery routine. However, according to the Informal Introduction to Algol 68,[‡] in some cases "a run-time check would probably be necessary, and we doubt whether most compilers will bother to put it in" (cf. §4.2.3 of [Lind69]). The consequence will be some very unusual program bugs.

To summarize, it appears that there is no completely satisfactory way to deal with static scoping of free variables while allowing procedures to have nonstatic scope. Further, dynamic scoping is simpler and in most cases more convenient for the programmer. Hence, we have chosen to use the dynamic scope rule in EL1.

[†]As usual, we have altered notation to avoid confusion. Algol 68 calls procedures "routines", and uses "procedure" meaning procedure-valued variable.

[‡]This is an informal description of Algol 68, prepared by C. H. Lindsey and S. V. van der Meuler at the request of the Algol 68 working group (IFIP WG 2.1).

7.2.3 Accessing the Values of Variables

In the specification of EL1 given in section 5, the value of a variable is always obtained by accessing the datum component of the symbol-table-element for that variable name. The binding and unbinding of variables at procedure entry and exit arranges that the datum component always yields the current incarnation of the variable name. There are, of course, a number of other techniques for implementing name scoping. In this sub-section, we will discuss one such alternative technique which presents itself as a particularly strong rival and examine the trade-offs between it and the one we have chosen.

It will be useful to first discuss in detail the method used in EL1. This uses three structures — the value-stack, the name-pdl, and the set of symbol-table-elements — which are employed as follows. (1) The value-stack is a lifo stack holding the actual values of all local variables. (2) The name-pdl is a stack of name-pdl-elements, one for each local variable, where

name_pdl_element ⇐ S(name : symbol, old_index : int, datum : ptr_any)

The datum component points to the actual value which resides in the value-stack. The old-index is the index of the name-pdl-element for the previous variable having the same name as this entry. The name-pdl is always kept current; hence, scanning the stack from top to bottom (i.e. most recent entry to oldest entry) the first entry encountered having a given variable name always corresponds to the current incarnation of that variable. (3) for each variable name, there is a symbol-table-element defined

```
symbol_table_element ← S(print_name : PTR(string),
                         datum : ptr_any,
                         pdl_position : int)
```

Let N be some variable name and let S_N be the symbol-table-element for N. The current incarnation of the variable named N is always pointed to by S_N . datum while S_N . pdl_position is the index of the corresponding element on the name-pdl. This is the same element that would be obtained by scanning the name-pdl from top to bottom looking for the name N. Hence, the current incarnation of N can be found either via S_N or by scanning the name-pdl. The evaluator for symbols used in section 5 — ev-symbol — uses the former.

Keeping S_N . datum current requires that whenever a variable named N is created, an entry E must be made on the name-pdl and the following actions taken

```
E . name ← N;
E . datum ← ⟨location of actual value on the value-stack⟩;
E . old_index ← S_N . pdl_position;
S_N . pdl_position ← ⟨index of E⟩;
S_N . datum ← E . datum;
```

On destroying the variable at procedure exit, it is necessary to reverse these actions

```
S_N . datum ← name_pdl[E . old_index] . datum;
S_N . pdl_position ← E . old_index;
```

Name scoping is implemented by a somewhat different technique in BBN Lisp 1.85 [Bobr68]. There, the symbol table entry is not used for the current value and the elements of the name-pdl[†] have the form

name_pdl_element ← S(name : symbol, datum : ptr_any)

The current value is obtained by searching the stack of these elements for the first occurrence of the desired name.

The advantage of the stack-only technique is its simplicity. It does not require setting up back pointers in the name-pdl, changing the symbol-table-elements, or unwinding the backpointers. Also, the absence of backpointers and slots to hold them result in less storage required by the name-pdl. Most important, however, is that it preserves earlier environments correctly. In any environment, it is possible to return to an enclosing environment by merely moving back the stack pointer for the name-pdl. Hence, it is possible to easily implement such language features as (1) a procedure return through several intervening procedure calls, and (2) an evaluator which while leaving control in the current environment evaluates a given form in an earlier environment. More generally, maintaining the environment as a simple stack proves to be a clean, well-chosen representation; it makes possible the saving of environments, the transfer between environments, and the manipulation of environments as data objects with very little overhead.

[†] In BBN Lisp, the structure is called the "pdl"; we use our terminology to obtain uniform notation.

The disadvantage of the stack-only technique is that each access to a variable in interpreted[†] code requires searching the name-pdl. If a variable is used with any frequency, this may become expensive. The search loop requires about 4 instructions, depending on machine. The most favorable assumption about variable usage is that all variables are local. Assuming that each procedure has around 5 locals[‡] this would imply that on the average 2 or 3 elements must be searched for the right one to be found. Hence, we can expect that individual accesses to a variable will take at least an order of magnitude longer than with the technique used by EL1. If the variable is used repeatedly, say in an iterative loop, this expense is particularly distasteful.

A second point should be noted. While most variables used in procedures are indeed local and their entries lie reasonably close to the top of the name-pdl, procedure names are an important exception: they are usually used free. Typically, a procedure name is bound at the outermost level (cf. §9.3 for a further discussion of this point). Because the name-pdl can grow large due to nested (particularly recursive) calls, considerable searching would be required to find their entries. Since procedure access is a frequent occurrence, the cost

---

[†] Compiled code "knows" the position on the name-pdl of all local variables. The locations of free variables can be determined by a search of the name-pdl on procedure entry and stored in name-pdl elements which are treated specially. The position of these special elements are also "known" to compiled code. Hence, once the locations of free variables have been picked up, a free variable can be accessed at about the same price as a local.

[‡] In Lisp terms, this corresponds to the number of λ-variables plus top-level prog-variables.

of such searching would be unacceptable. In section 7.1.5, we have discussed the means by which BBN Lisp escapes from this bind: procedure values are treated as a special case. A procedure value for a name N is stored in a special function-cell in the symbol table entry for N and this cell is accessed when a procedure value is needed (i.e. for procedure application). Hence, procedure values in BBN Lisp are accessed in the same way as are all values in EL1. There is, however, one difference. The function-cell in Lisp 1.85 is strictly global; it is not updated for local variables. The function-cell is always considered first when evaluating a name appearing as the operator in a procedure application; hence, a global value of some given name will sometimes override a local procedure value of that name. Although Bobrow [Bobr69] contends that this is precisely the appropriate scope rule for procedure names, we argue that it is more accurately seen as a language anomaly.

To summarize, the stack-only technique used by BBN Lisp has two difficulties. (1) Procedure values are either handled on the stack, in which case access is unacceptably slow, or they are handled specially, in which case their scope rules are non-standard. Neither, we contend, is acceptable. (2) Other values, while not so deeply burried, still require an order of magnitude more time to access than with the EL1 technique. On the other hand, the EL1 technique requires more set-up time when a variable is created or destroyed, requires more storage for the name-pdl, and makes manipulation of the environment more difficult.

The last difficulty is partly mitigated by the observation that the ELl technique does keep a valid stack. Hence, environments can be saved by copying the stack (or by switching the stack pointer which addresses it) and restored by reinstating the stack and then using it to reconstruct the values in the symbol table entries. Further, since the bindings of variables can be found by searching the stack, it is possible to temporarily use an enclosing environment by moving the name-pdl pointer and setting a switch so that ev-symbol does a search rather than using the datum cell. Provided that this is not done frequently, and infrequent use seems likely, this will be a satisfactory solution.

We chose the technique employed in section 5 for two principal reasons. (1) The special scope rule for procedure names seems very undesirable. (2) The expense of access to variables by stack search would be particularly painful since initially no compiler will be available.

It should be stressed, however, that this is an implementation issue to which the language is, or should be, insensitive. Hence, an implementation is free to replace the modules which handle name scoping with any others which produce the same results. When a running implementation is available, we intend to perform measurements on system behavior which will provide data to replace our guesses on such quantities as relative frequency and scope of procedure names, and average depth of stack searching which would be required in a stack-only scheme. Based on such data, we intend to reconsider the issue in a more enlightened fashion.

7.3 THE FORMAL DEFINITION

There are a number of criteria which might be applied in assessing the worth of a formal semantic specification and comparing it to others. The principal ones are:

(1) generality and power of the metalanguage employed,

(2) precision and range of the formal specification,

(3) directness and clarity of the specification,

(4) utility of the specification,

(5) independence of the specification from machine and implementation.

These are not all of equal importance. We will consider them in turn, first examining the criterion itself and then applying it in assessing the semantic specification of EL1.

In discussing the power and generality of the metalanguage, there are two different questions one might ask. (1) How large a semantic space does it span? (2) How large a space does it span effectively? The first turns out to be a pseudoquestion: almost every metalanguage is universal and will describe any programming language feature whatever. The issue of effective spanning is, however, a real one. It is necessary that the metalanguage effectively cover the programming language, but also that it cover equally well a large peripheral region in which metaphrase extensions can flourish. In this regard, EL1 as a metalanguage is clearly superior to austere formalisms such as the λ-calculus used by Landin, the variant of Markov algorithms employed by van Wijngaarden, or even the state vector model of McCarthy (cf. §2.1). It is roughly comparable to ULD (cf. §2.1.4) in the sorts of structures and operations it permits. However, EL1 permits one very useful information structure prohibited in ULD: shared components. It will be recalled that ULD structures are restricted to

trees; re-entrant graphs are forbidden. Hence sharing, e.g. of an object among two or more names, must be indirectly modeled in ULD whereas there is a direct representation in EL1 by means of a pointer. In this respect, EL1 is a significantly better metalanguage.

The precision of the formal specification turns out to be far less important an issue than it might first appear. All the formalisms that have been proposed for semantic modeling – from the λ-calculus to ULD – are mechanical. As such, they all yield a definite result for each program (possibly a set of results, in the case of ULD) with none of the ambiguities and vagueness which result from a natural language description. There is little reason to claim one model superior to another because it is more precise or is based on a "better" axiomatized metalanguage.

Semantic specifications do, however, differ in their range, i.e., how far they go in describing the language. Two dimensions are relevant:

(1) depth – that is, how far the reduction toward elementary operations proceeds; e.g., is the addition of integers formally defined?

(2) breadth – that is, how much of the environment in which the language runs is described in the specification; e.g., is the file system included? the garbage collector?

Some authors such as J. DeBakker [DeBak67] believe an extensive range to be a significant criterion. Were this indeed the case, then the specification of EL1 given in section 5 would necessarily be judged somewhat deficient. It leaves as primitives, unspecified by formal definition, the basic arithmetic operators, some mode creation operators, and several others; similarly, it does not treat a number of system features such as garbage collection and input/output.

However, it is our contention that once a certain domain has been covered, an extensive range is relatively unimportant. For example, in the case of EL1, it would not have been difficult to continue the formal definition in both dimensions. The mode INT could be defined as ROW(k, BOOL) and arithmetic operations defined by procedures carrying out binary arithmetic modulo k. However, it is unclear what purpose such an exercise would serve. The arithmetic operations over the integers are well defined; there is no question of ambiguity or uncertainty as to the result. Similarly, input/output devices and input/output primitives could be constructed in the language instead of being left primitive. Again, it is not at all clear that such a continuation would be either useful or interesting in the context of this study. In general, there are many ways of defining the semantics of some component of a language; formal specification is only one of many. In the language proper, a formal definition serves well, for it gives precision in the region where the traffic and interaction of components are most heavy. At the peripheries, operations are isolated and less complex; the same degree of precision is unnecessary. In short, while a formal semantic specification is most appropriate in describing the kernel language, attempts to apply it outside this province may be misplaced.

One criterion we do believe to be valid is the directness and clarity of the specification. Obviously, this is a prerequisite if the specification is to be read and used. Less obvious perhaps is its importance in obtaining a correct specification. Formal specifications, like programs, must be debugged. The definition of any nontrivial language, in any metalanguage, will be sufficiently complex that it may be <u>expected</u> to contain errors. Some will be mere misprints, but one or more oversights should not be

surprising. The syntax of Algol 60 was a far simpler matter, examined by the several members of the defining committee, yet it contained a serious ambiguity. Oversights of similar importance should be expected in a semantic specification, initially. If a semantic specification can be circulated, read, studied, and understood by a community other than its authors, the work can be effectively refereed and errors discovered. Failing such exposure, the correctness of the work may remain in doubt.

In attempting to apply the criterion of directness and clarity, it must be confessed that this is partly a matter of taste. Issues such as style enter in to some extent. However, the limiting and most important factors tend to be the model employed. We noted in section 2.1 the advantage of a one-stage, interpreter-based formal model in giving a direct specification of languages, specifically of establishing a one-to-one correspondence between structure and meaning. Also, we have noted the utility of making a semantic specification intuitively acceptable by using an effective, straightforward representation. Examining the specification given in section 5, it appears that these guidelines have served well. Considering the power of the language, its definition is surprisingly small, clean, and perspicuous.

Another valid criterion is the utility of the formal definition. One may well ask for what purposes the definition was intended and how well these purposes are served. Three classes of uses seem most important:

(1) communication – to users, students, standards committees, and the like,

(2) as a design tool,

(3) as an implementation guide.

To these three, some schools might add a fourth:

(4) as a basis for proofs about the language.

Before discussing these, it should first be pointed out that many formal specifications of programming languages are intended for no use whatever. Two of the larger efforts – DeBakker's model of Algol 60 and the ULD model of PL/I – appear to be exercises in formal definition unalloyed with any thought of use. The ULD project, for example, was started too late to be employed in the design of PL/I, in assessing proposed changes to the language, or as a guide to the IBM System/360 implementation. There was once some toying with the idea of using the ULD specification as a "language control document", to authoritatively define the language [Nich68]; however, nothing has come of this. In short, a useful formal specification is the exception rather than the rule. This, more than any other single observation, indicates the immature state of the field.

The formal specification of EL1 is intended to serve the first three uses cited above: communication, design, and implementation. We recognize the potential importance of the fourth but find it outside the range of the present study. At this state in the development of the field, it would appear that the construction of formal proofs is less important than the ability to carry out clear, coherent discourse at the descriptive level.

The utility of the EL1 formal definition in communication can only be judged by others; we leave this assessment to the reader. We can, however, comment on its application in design. There it proved invaluable. We used the formal specification as our working notation: instead of drawing diagrams with tangles of pointers in the usual fashion of system programming, we wrote abstract syntax and EL1 interpreter code. Since EL1 was designed partly with this use in mind, it should hardly be surprising that it proved to be a facile and convenient notation. However, this does support the extensible language thesis: the right language is the one

tailored to the task. In particular, the availability of the data type definition facility for constructing the abstract syntax of programs and the structures used by the evaluators was of singular importance.

As EL1 has not yet been implemented, the acid test of the formal specification is still to come. We wrote the specification with the intention that it be used as a blueprint for implementation. Further, in designing the language, we made many design decisions based on the model; for example, judging the efficiency of a construct from its treatment in the model. This raises certain questions which bear examination. If such judgments are to produce an efficient language, it is necessary that the semantic specification be as realistic as possible, i.e., that it serve as a plausible model for pragmatics as well as a reliable model of semantics. While the model need not display every detail of the actual processor, we want some assurance that simple actions in the model can be carried out efficiently in practice. This imposes two requirements. One has already been noted in our discussion of clarity of specification: primitives of the model must be appropriate to reasonable contemporary hardware wherever possible. This simultaneously insures that the model is intuitively acceptable and pragmatically valid at the base level. The second requirement is that the structures used in the specification be homologous to those which are intended for use in the implementation. Note that while the first requirement is not difficult to accept, the second may present problems: wherever the implementation is to use an efficient but complex structure instead of a simpler, less efficient one, there will be question as to whether or not the model should use the simpler one instead. While the temptation may be present, it is our contention that such an attempt at simplification would be misspent. As it creates a purely artificial

processor, it will make comprehension of the language and its specification more difficult, for valid choices made for good reasons appear mere whimsey when projected onto an artificial model irrelevant to the actual design.

In one sense, EL1's use of semantic specification as an implementation blueprint is a rejection of the notion of implementation independence, at least in its strongest interpretation. Unlike ULD, which attempts to describe PL/I in a fashion independent of any possible implementation, our formal specification yields a preferred implementation for EL1. While any other implementation which produces the same results is equally valid, there is a strong predisposition toward the implied implementation. Note that the issue here is quite distinct from machine independence. Most models proposed for semantic specification, including our own, do not depend on a specific computing machine, IMP being the only notable exception. By implementation and dependence thereon, we refer to the structures and tables employed, the processor modules and their relation, and such like – a level of organization above the actual machine. There are those who argue that implementation independence is a highly desirable goal, for the semantic specification is then "pure" and therefore in some sense "better". Taken strictly as a validity rule, implementation independence is an unassailable principle: all processors which produce the results specified by the formal definition are surely equally valid. However, we contend that it would be folly to take this as a normative rule and choose a specification which is unimplementable and hence implementation-independent. Quite the contrary, as pointed out above, there are good reasons to design a language with a specific implementation in mind and embody this as a paradigm in the formal specification.

Section 8. EXTENSIONS

The first thing one does with an extensible language is extend it. In this section, we specify a number of extensions serving a variety of functions. Some are designed to mimic features which have proved useful in other languages. Were a handbook of language features ever compiled, it would serve as a source of scenarios; in the absence of such a compendium, we shall raid user's manuals. A second set of extensions is concerned with establishing a facility with which higher-level mode definitions can be constructed.

Concerning the specification of these extensions, it should be recalled that EL1 is at present only a base language and does not contain a complete extension mechanism. Consequently, the extensions will be presented partly as modifications to the specification of section 5 and only partly as executable statements in the language.

8.1 LISTS, PROPERTY LISTS, AND LIST PROCESSING

A survey of the literature discloses that the sine qua non of an extensible language proposal is a demonstration of the ability to perform list processing, preferably in the spirit of, and using the notation of Lisp. To mimic Lisp 1.0 [McCar60], the necessary mode definitions are straightforward:

```
DECL dotted_pair, list : mode;
dotted_pair ← allocate(ddb, ⟨ ⟩);
list ⇐ PTR(dotted_pair, symbol_table_element);
dotted_pair ⇐ S(car:list, cdr:list);
```

Similarly, the primitive operations are readily defined. For example,

```
cons ←
PROC(x:list, y:list) list;
DECL temp:list;
temp ← allocate(dotted_pair, ⟨ ⟩);
temp.car ← x; temp.cdr ← y;
temp ENDP;

car ←
PROC(x:list) list; x.car ENDP;

atom ←
PROC(x:list) list;
mval(x) = symbol_table_element ⇒ t;
ELSE f;
ENDP;
```

where "t" and "f" are identifiers of mode list whose values are pointers to distinguished symbol table elements. The other primitive operations — cdr and eq — are defined analogously. From these, the various list manipulation functions may be obtained by transcribing the Lisp definitions into EL1. For example,

```
subst ←
PROC(x:list, y:list, z:list) list;
NT This substitutes the list x for the atom y in the list z;
atom(z) ⇒ [[ eq(z,y) ⇒ x; ELSE z ]];
ELSE cons(subst(x,y,car(z)), subst(x,y,cdr(z)));
ENDP;
```

If, as will surely be the case, one is interested in the list processing of Lisp 1.5 rather than Lisp 1.0, two sets of additions must be made. The first is simple: to allow previously created list structure to be changed. We define

```
rplaca ←
PROC(x: list, y: list) list;
mval(x) ≠ dotted_pair ⇒ error("rplaca_fault");
x.car ← y;
x ENDP;
```

The function rplacd is similar. The second addition is to allow atoms to be integers as well as symbol table elements. We replace the above definition for list by

```
list ⇐ PTR(dotted_pair, symbol_table_element, INT);
```

and modify the definition of atom. This, however, is incomplete for while it allows a list to be a PTR(INT), it does not allow a list to be an INT. Hence, forms such as

```
cons(x, 3+5)
```

are not legal, for the second argument to cons is an INT. The remedy is simple: the primitive operations are changed to take generic arguments, i.e., list or INT. When necessary, the INT is converted to a PTR(INT). We define the mode

```
int_or_list ⇐ RANY(INT, list);
```

and the conversion function

```
int_to_list ←
PROC(x:INT) list;
DECL temp:list;
temp ← allocate(INT, ⟨ ⟩);
val(temp) ← x;
temp ENDP;
```

Then the definition of cons becomes

```
PROC(x:int_or_list, y:int_or_list) list;
DECL temp:list;
temp ← allocate(dotted_pair, ⟨ ⟩);
temp.car ← [[typ(x) = INT ⇒ int_to_list(x); ELSE x]]
temp.cdr ← [[typ(y) = INT ⇒ int_to_list(y); ELSE y]]
temp ENDP;
```

This and analogous changes to eq, rplaca and rplacd are the only modifications necessary, for all the other functions are defined in terms of the primitives.

It should be clear, however, that procedures which are given or deliver objects of the wrong mode will be a common problem in an extensible language due to the large number of defined modes. The more syncretistic the definition set, the greater the problem. In section 8.4, we discuss a general solution.

One feature of Lisp which makes it an attractive programming language is the property list which allows each symbol to have arbitrary information associated with it under programmer control. This serves as a convenient means of storing data to be retrieved using that symbol as a key, e.g. relations, dictionaries, alternative procedure definitions, and

other attributes. Given the data type list and list processing operations, addition of a property list to EL1 is simple. It is necessary only to redefine the mode symbol-table-element to include a component which holds the property list (i.e., "proplist").

```
symbol_table_element ⇐ S(print_name: PTR(string),
                         datum: ptr_any,
                         pdl_position: int,
                         proplist: list);
```

The functions for searching, adding, and removing properties from a proplist are as in Lisp 1.5.

Another exercise is provided by the use of three-link cells:

```
triple ← allocate(ddb, ⟨ ⟩);
list3 ⇐ PTR(triple, symbol_table_element, INT)
triple ⇐ S(car: list3, cdr: list3, csr: list3);
```

Doubly linked lists may be formed by chains of triples in which the cdr cell points to the next triple and the csr cell points to the previous triple. That is, if x is a list3 pointing to such a chain,

```
x.cdr.csr = x
x.csr.cdr = x
```

except where the cdr or csr is NIL. Such a chain may be readily traversed in either direction. For example, consider

```
nth ←
PROC(x:list3 BYVALUE, n:INT) list3;
DECL s:symbol;
[[n > 0 ⇒ s ← "cdr"; n < 0 ⇒ s ← "csr" ]]
FOR i ← 1, . . ., abs(n) WHILE x ≠ NIL DO x ← x[s];
x ENDP;
```

This yields the list obtained by moving |n| positions – forward if n is greater than 0 or backward if n is less than 0.

Other uses for three-link cells include doubly-linked rings and binary trees with contents cells. Further, by adding a component to hold a BOOL flag, we can obtain the threaded lists of Perlis and Thornton [Perlis60].

8.2 SEQUENCING CONSTRUCTIONS

One weakness in the surface structure of EL1 is its paucity of control structures. Without going into the deeper issues of control, we observe that there are a number of possible special-purpose forms for expressing conditional evaluation, selection of evaluated forms, and similar sequencing rules. One obvious candidate is a two-armed conditional form, with concrete syntax

form → IF form THEN form ELSE form

This is distinct from another obvious addition, the one-armed conditional with concrete syntax

form → IFF form THEN form

By using two distinct constructs, we make parsing easier for the analyzer as well as the human reader. Both forms may be mapped directly into the abstract syntax type compound-form, so that neither special evaluator nor additional abstract syntax is required.

A more interesting example is the case construction of Wirth and Hoare [Wir66b]. This permits the selection and evaluation of one form from a set of forms, the selection being made in accordance with the values of an integer switch expression. For example, consider

```
CASE f(x) IN
    a ← delete(b,c);
    [[ p(x) ⇒ s, ELSE s ← t & w ]];
    g(b[j] ← c);
    ENDC
```

This evaluates f(x) and then executes either the first, second, or third form between IN and ENDC , depending on whether the value of f(x) is 1, 2, or 3. (If f(x) is not an INT or does not fall between 1 and 3, then the last form is taken by default.) The value of the case form is the value of the selected form.

The precise definition is as follows:

(1) concrete syntax

$$\text{form} \rightarrow \text{CASE form IN } \{\text{form};\}^{\oplus} \text{ ENDC}$$

(2) abstract syntax

```
case ⇐ S(switch:form, body:formρ);
```

(3) evaluator

```
evcase ←
PROC(c:case) ptr_any;
DECL i:int;
DECL p : ptr_any;
```

```
DECL last: int;
last ← length(c.body);
p ← eval(c.switch);
mval(p) ≠ INT ⇒ eval(c.body[last]);
i ← val(p);
(1 ≤ i) ∧ (i ≤ n) ⇒ eval(c.body[i]);
ELSE eval(c.body[last]);
ENDP;
```

The choice to evaluate the last form in the case when the switch is not an INT between 1 and n is purely a matter of taste. Our belief is that this will prove most convenient. However, plausible arguments can be made for ruling this an error, or providing an explicit default form, e.g.,

form → CASE form IN $\{\text{form};\}^{\oplus}$ DEFAULT form

with abstract syntax

```
case ⇐ S(switch: form, body: formρ; default: form);
```

and appropriately modified evaluator.

8.3 SEPARATE FUNCTION CELL

As discussed in section 7.1.5, it may be convenient to allow a procedure to be associated with an identifier quite independently of any other values which that identifier denotes. Thus, for example, "transform" could name a procedure simultaneously with its use as a formal parameter, say of mode INT. Context determines which of the two values is intended: the procedure is implied only where the symbol "transform" appears as a binary operator or as a procedure name in a procedure-application; the other value is implied in all other circumstances.

A definition set to allow this, using the scope rules of BBN Lisp as discussed in section 7.1.5, is as follows.

(1) The abstract syntax for symbol-table-element is redefined:

```
symbol_table_element ⇐ S(print_name: PTR(string),
                         datum: ptr_any,
                         pdl_position : int,
                         proplist : list,
                         fn_cell : proc_var);
```

(2) To set the function cell, we need a special procedure

```
putd ←
PROC(x: form UNEVALED, y: proc_var) proc_var;
mval(x) ≠ symbol ⇒ error("type_error");
val(val(x)) . fn_cell ← y;
ENDP
```

The procedure, getd, for explicitly accessing the function cell is analogous.

(3) The evaluator, ev-proc, is defined as follows:

```
ev_proc ←
PROC(p: proc_form) ptr_any;
DECL s: symbol;
NT If p is neither a symbol nor a pointer to a symbol, then
    ordinary evaluation takes place;
(typ(p) ≠ symbol) ∧ (mval(p) ≠ symbol) ⇒ eval(p);
s ← [[type(p) = symbol ⇒ p; val(p)]];
NT Try fn_cell;
s.fn_cell ≠ NIL ⇒ s.fn_cell;
```

```
        NT  If fn_cell empty, use ordinary evaluator for symbols;
        eval_symbol(s);
        ENDP;
```

where

```
        proc_form ⇐ RANY(symbol, form);
```

(4) In the evaluator, the code must be changed to call ev-proc when a procedure is to be evaluated. This occurs twice in ev-binary-op and once in apply. For example, the corrected apply reads

```
        apply ←
        PROC(f: procedure_application) ptr_any;
        apply2(check_proc ∘ ev_proc(f.operator), f.arguments);
        ENDP;
```

8.4 PROGRAMMER-DEFINED SELECTION, ASSIGNMENT, AND CONVERSION FUNCTIONS

In section 7.1.1, we noted that while the sorts of basic objects allowed in EL1 are somewhat restricted, a far wider domain can be obtained through higher-level data type definitions. Here, we present a set of metaphrase extensions which make possible such definitions. We will refer to the extension set as the extended mode definition facility.

The technique is motivated by the observation that a data type definition is, or rather should be, a description of its behavioral laws. That is, a data type is completely described by specifying what sorts of data its instances can contain and how those data are stored into and retrieved from an instance. At this level of discourse, storage formats and the like are merely implementations of the axiomatic behavioral laws.

Using the basic objects as building blocks, higher-level data types can be obtained by defining functions which describe the desired behaviors. For example, a list may be represented by linking together dotted-pairs; however, it is frequently useful to treat a list x as if it were a single object and, for example, denote its i^{th} element by "x[i]". The meaning of this construct is specified by a programmer-defined selection function associated with the mode list. Typically, the desired definition will be†

```
PROC(a: list BYVALUE, n: INT) list;
FOR j ← 1, . . ., n - 1 DO a ← cdr(a);
car(a) ENDP;
```

In general, an extended mode definition is obtained by taking an ordinary mode and associating with it three programmer-defined functions: select-fn, assign-fn, and convert-fn. The first two correspond directly to the system-defined selection and assignment functions. The third performs conversion from the defined mode to other modes as required. The use and relation of these three may be best presented by means of an extended example.

Suppose we wish to define a class of lifos which hold only integers and which can be used in ordinary arithmetic statements. For example, if x is such a lifo,

```
x ← 3 * i
```

pushes the value of 3i onto x, and

```
x + 2 * j
```

---

†Note that this definition has the property that subscripted lists may appear to the left of the assignment operator, e.g., "x[i] ← y" has the desired result.

pops the top value from x and adds it to 2j. Further, it is convenient to allow access, without removal, to elements of x other than the top, e.g.,

```
k + x[j]
```

adds to k the j^{th} element of x, provided that x holds at least j elements. The definition for the base mode is

```
int_lifo ⇐ S(index: int, body: intρ)
```

so that x could be declared an int-lifo by

```
DECL x: int_lifo SIZE ⟨n⟩;
```

The desired extended mode is constructed by augmenting the base mode int-lifo with three programmer-defined functions. We first consider a select-fn. By a mechanism to be discussed later, this select-fn will be called to evaluate all selections on which the object being selected is an int-lifo, e.g., "x[i+3]"

```
PROC(x: int_lifo BYREF, i: int) int;
(1 ≤ i) ∧ (i ≤ x@index) ⇒ x@body[i];
ELSE error("select_int_lifo");
ENDP;
```

This checks that i is between 1 and the index and, if so, performs the appropriate subscripting of the body which is an intρ; otherwise, error is called. The notation "@" requires some explanation. It will be recalled (cf. §5.10) that a selection is written either in the format

```
form2 . identifier
```

or

```
form2 [form]
```

where the former is syntactic sugar for

form2 ["identifier"]

In line 2 of the function, it is necessary to refer to the index component of the int-lifo x. Normally, one would denote this by "x.index". However, the above programmer-defined select-fn is to be applied to evaluate all selections whose left part is an int-lifo. Hence, using "x.index" in the above function would invoke an erroneous recursive call on the function itself. What we want is to take the index component of x as defined by the base mode definition for int-lifo. That is, the higher-level select-fn must refer to the primitive representation used in the base mode. The symbol "@" is introduced to specify that the primitive representation is called for. There are two selection formats specifying primitive representation

form2 @ identifier

and

form2 @ [form]

where, again, the former is syntactic sugar for

form2 @ ["identifier"]

The rule for evaluating an arbitrary selection can now be stated.

(1) The object to be selected is evaluated. Let its mode be $\mathfrak{M}$.

(2) If $\mathfrak{M}$ has a programmer-defined select-fn and if primitive representation is not called for in the selection, then the programmer-defined select-fn is applied to the object and its field.

(3) Otherwise, the basic selection function for the mode $\mathfrak{M}$, found in $\mathfrak{M}$. s_fn (cf. §5.9.4), is applied.

No restriction is placed on the sort of field which a programmer select-fn may take as its second argument. In the above example, the second argument, i, was an int because it is convenient to perform selection on

int-lifos using an int field. For other modes, various other selection fields may prove more convenient, e.g.,

```
u[⟨i+j, i-j, k⟩]        (here the field is an intρ of length 3)

v['a]                   (here the field is a CHAR)

w.ordinary              (here the field is a symbol)

z[⟨complex: x, y⟩]      (here the field is a complex number)
```

The changes to language specifications needed to make this work are as follows.

(1) The mode ddb (cf. §5.9.4) is redefined to contain slots† for holding a programmer-defined select-fn, an assign-fn, and a convert-fn. (The latter two will be needed later.)

```
ddb ⇐ S(d: type_descriptor,
        class: symbol,
        type_resolved: bool,
        dope_length: int;
        a_fn: proc_var,
        s_fn: proc_var,
        canonical_name: symbol,
        select_fn: proc_var,
        assign_fn: proc_var,
        convert_fn: proc_var);
```

---

†Since these three slots will not always be occupied, it might prove useful to use a property list instead of reserving slots. However, note that this will effect a saving only when the average number of programmer-defined functions per mode is considerably less than 1.

(2) The concrete syntax for selection (cf. §5.10.1) is redefined

```
selection → form2 field
field → . identifier | [ form ] | @ identifier | @ [ form ]
```

(3) The abstract syntax is correspondingly augmented

```
selection ⇐ S(object:form, field:form, primitive_flag:bool);
```

On converting from parse tree to abstract form, the primitive-flag is set TRUE if the concrete field has the format "@ identifier" or "@ [form] ".

(4) The evaluator of selections, ev-selection, is redefined so that it applies the programmer-defined select-fn when appropriate. The new definition is

```
ev_selection ←
PROC(s:selection) ptr_any;
DECL x, result:ptr_any;
DECL saved_flag:bool;
DECL m:mode;
x ← eval(s.object);
m ← mval(x);
(m.select_fn ≠ NIL) ∧ (s.primitive_flag = FALSE) ⇒
    apply2(val(m.select_fn), ⟨formρ:s.object, s.field⟩, x);
x ← dereference(x);
[[pure_value(x) ⇒ BEGIN x ← save(x); saved_flag ← TRUE END]];
result ← select2(x, field_index(m, s.field));
```

```
[[ saved_flag ⇒
    BEGIN
    result ← return_result(result, mval(result));
    free_last(value_stack);
    END ]];
result ENDP;
```

(5) One further set of changes is required. To determine whether a programmer select-fn is to be used (and if so, what function), it is necessary for ev-selection to evaluate the object, so as to obtain its mode. If a select-fn is in fact to be used, we wish to call apply2 which will carry out the function-application. However, there is one difficulty: apply2 was defined to take the arguments of the function-application as a set of unevaluated forms and evaluate these during binding (if required). Normally, this would include the first argument to the function-application: the object. Note that this has already been evaluated. It is undesirable to evaluate the object a second time, particularly in view of the expected frequency of repeated selection (e.g., "b.sel[j].foo[f(x)]") as well as the possibility of side effects. Hence, the evaluated object is passed to apply2 as a third argument. This requires the procedure heading of apply2 to be changed to

```
PROC(p: procedure_block, args: formρ, arg1: ptr_any) ptr_any;
```

Corresponding changes must be made in all calls on apply2. In the binding of formal parameters to their arguments, if arg1 is present (i.e., non-NIL) then it is used in place of evaluating arg[1]. This requires changing line 19 of bind-formals (§5.13.4) to

```
arg ← [[ (i = 1) ∧ (arg1 ≠ NIL) ⇒ arg1; ELSE eval(args[i]) ]];
```

Continuing with the example of int-lifo, we next consider the programmer-defined assign-fn. This, it will be recalled, is to allow forms such as

```
x ← 5*j
```

where x is an int-lifo, meaning: push the value of 5j onto x. A possible definition is

```
int_lifo.assign_fn ←
PROC(x: int_lifo BYREF, y: INT) INT;
x@index = length(x@body) ⇒ error("int_lifo_overflow");
x@body[x@index ← x@index+1] ← y;
ENDP;
```

As with the operation of selection, it is sometimes necessary to perform assignment using the primitive representation. To override the programmer-defined assign-fn and invoke the assignment function of the base mode, the operator ":=" is used. For example, if x and y are both int-lifos,

```
x := y
```

is an assignment of int-lifos† and is equivalent to

```
[[x@index ← y@index; x@body ← y@body]]
```

In general, the rule for performing assignment is similar to that for selection. The left-hand operand is evaluated. If its mode has a programmer assign-fn and if primitive representation is not called for (i.e., the assignment symbol is "←"), then the assign-fn is applied. Otherwise, the base

---

†When the definition set is complete, it will be seen that the related form "x ← y" is an assignment of ints.

mode assignment function is applied.

To put this evaluation rule into the language, only two procedures of the evaluator – ev-binary-op and assign – need be changed. The syntax, concrete and abstract, remains the same.

```
ev_binary_op ←
PROC(b: binary_operation) ptr_any;
(b.op = "←") ∨ (b.op = ":=") ⇒ assign(b);
b.op = "∘" ⇒ apply2(checkproc∘eval(b.lhs), ⟨formρ: b.rhs⟩, NIL);
ELSE apply2(checkproc∘eval_symbol(b.op), ⟨formρ: b.lhs, b.rhs⟩, NIL);
ENDP;

assign ←
PROC(b: binary_operation) ptr_any;
DECL left, right: ptr_any;
DECL pv_flag: bool;
DECL m: mode;
left ← eval(lhs);
m ← mval(left);
(m.assign_fn ≠ NIL) ∧ (b.op = "←") ⇒
    apply2(val(m.assign_fn), ⟨formρ: b.lhs, b.rhs⟩, left);
[[pure_value(left) ⇒ BEGIN left ← NIL; pv_flag ← TRUE END]];
right ← eval(rhs);
pv_flag = FALSE ⇒ assign2(left, right);
right ENDP;
```

The third sort of programmer-defined functions used in constructing extended mode definitions is the convert-fn. To continue our example, we

observe that it is often desirable to use an int-lifo x in an argument position where an int is required, e.g.,

```
x + 3
```

meaning that x is to be popped and the popped element is to be added to 3. When the evaluator has in hand an int-lifo but a value of different mode is required, the convert-fn for int-lifo is called with two arguments: (1) the int-lifo, (2) the mode of the value required. A possible definition is

```
int_lifo.convert_fn ←
PROC(x: lifo BYREF, m: mode) m;
DECL temp: INT;
m = INT ⇒
    [[ x@ index < 1 ⇒ error("int_lifo_underflow");
     temp ← x@ body[x@ index];
     x@ index ← x@ index - 1;
     temp ]];
m = bool ⇒
    [[ temp ← int_lifo.convert_fn(x, INT);
     temp = 0 ⇒ FALSE;
     ELSE TRUE ]];
ELSE error("lifo_convert_error");
ENDP;
```

This allows an int-lifo to be used either as an int or a bool. In the latter case, we obtain an int and convert this to a bool. With appropriate changes, we could provide for supplying a complex number, a string, a quaternion, or any other sort of object which might be required.

In general, a convert-fn for mode $\mathfrak{M}$ is called with two arguments: (1) an object of mode $\mathfrak{M}$, (2) the expected mode $\mathfrak{M}'$. The result of the convert-fn will either be an $\mathfrak{M}'$, in which case the conversion is complete, or some $\mathfrak{M}''$ not equal to $\mathfrak{M}'$, in which case the convert-fn for $\mathfrak{M}''$ will be called. For example, the conversion

bool → complex

might be carried out in two stages

bool → int int → complex

using the convert-fn for bool, followed by the convert-fn for int.

In the scheme we propose, automatic conversions via the convert-fns are carried out under two circumstances:

(1) when binding arguments to formal parameters (e.g., to allow "x+3" when x is an int-lifo), in which case the expected mode is the declared type of the corresponding formal parameter (cf. §5.12.3),
(2) when exiting a procedure (e.g., to allow a procedure whose declared mode is int to return an int-lifo), in which case the expected mode is the declared type of the procedure (cf. §5.13.4).

Implementing this requires changing bind-formals and proc-exit (cf. §5.13) so that they call a new procedure, convert.

(1) In bind-formals, lines 20 and 21 are replaced by

```
arg ← convert(m, arg);
```

(2) In proc-exit, the first argument is called BYVALUE and lines 3 through 7 are replaced with

```
result ← ⟦expected_mode = none ⇒ NIL;
          ELSE convert(expected_mode, result)⟧;
```

Convert decides whether a result is compatible with the mode required for that result and, if not, whether there is a programmer-defined convert-fn

which can be applied to perform conversion. Convert is defined:

```
convert ←
PROC(expected_mode: mode BYREF, result: ptr_any) ptr_any;
DECL result_mode: mode;
result_mode ← mval(result);
(result_mode = expected_mode) ∨
    compatible(expected_mode, result_mode) ⇒ result;
(expected_mode.class = "rany") ∧
    alternative(result_mode, expected_mode) ⇒
    [[ expected_mode ← result_mode; result ]];
result_mode.convert_fn ≠ NIL ⇒
    convert(expected_mode,
            apply_convert_fn(val(result_mode.convert_fn),
                             result, expected_mode));
ELSE error("convert_error");
ENDP;
```

This uses two auxiliary routines, alternative and apply-convert-fn. The former is defined:

```
alternative ←
PROC(r: mode, u: mode) bool;
NT  This returns TRUE iff r is an alternative of u;
DECL found_flag: bool;
FOR i ← 1, . . ., length(u.d) TILL found_flag DO
    [[ u.d[i] = r ⇒ found_flag ← TRUE ]]
found_flag ENDP;
```

The latter requires a mode definition

```
either_formal ⇐ RANY(expr_formal, code_formal);
```

It is defined:

```
apply_convert_fn ←
PROC(p: procedure_block, value: ptr_any, expected_mode: mode)ptr_any;
NT  This applies p to the two arguments value and expected_mode;
DECL fm, declared_type: mode;
DECL temp, result: ptr_any;
DECL f: either_formal SPECIF
    [[type(p) = explicit_procedure ⇒ expr_formal; ELSE code_formal]];
length(p.formals) ≠ 2 ⇒ error("convert_error");
NT  Bind first argument;
f ← p.formals[1];
fm ← [[type(f) = code_formal ⇒ f.type;
        ELSE eval_to_type(f.type, mode)]];
not∘compatible(fm, mval(value)) ⇒ error("convert_error");
f.bind_class = "UNEVALED" ⇒ error("convert_error");
BEGIN
    (f.bind_class = "BYREF") ∧ not∘pure_value(value) ⇒
        install_variable(f.name, NIL, ⟨ ⟩, value);
    temp ← install_variable(f.name, fm, dope_vector(value), NIL);
    assign2(temp, value);
END;
```

```
NT  Bind second argument;
f ← p.formals[2];
fm ← [[ type(f) = code_formal ⇒ f.type;
        ELSE eval_to_type(f.type, mode) ]];
fm ≠ mode ⇒ error("convert_error");
temp ← install_variable(f.name, mode, ⟨ ⟩, NIL);
val(temp) ← expected_mode;
make_current(2);
NT  Determine the declared type;
declared_type ← eval_to_type(p.result_type, mode);
NT  Evaluate the procedure;
[[ type(f) = code_procedure ⇒
     result ← xct(p.body, name_pdl, pdl_index);
  ev_declarations(p.declarations);
  result ← ev_statementρ(p.statements) ]];
NT  Clean up and exit;
result ← proc_exit(result, declared_type, pdl_index - 2);
pdl_index ← pdl_index - 2;
result ENDP;
```

By defining a single convert-fn, it is possible to correct a class of type mismatches globally. Instead of writing or changing a set of procedures to accept generic arguments, a single convert-fn can be used. This is particularly useful in view of (1) possible changes to the desired method of conversion, and (2) possible additions to the set of affected procedures. For example, returning to linked lists discussed in section 8.1, we can allow forms such as

```
cons(x, 3+5)
```

or

```
rplaca(y, 6)
```

by defining the convert-fn for int to deliver a list when required:

```
int.convert_fn ←
PROC(i: INT, m: mode) m;
NT  This handles conversion of ints to lists, bools, or complex numbers;
DECL temp: list;
m = list ⇒
    ⟦temp ← allocate(int, ⟨ ⟩); val(temp) ← i; temp⟧
m = bool ⇒ ⟦i = 0 ⇒ FALSE; TRUE⟧;
m = complex ⇒ ⟨complex: i, 0⟩;
ELSE error("int_convert_error");
ENDP;
```

Hence, whenever a list is needed and an int is held in hand, the appropriate conversion will take place.

Section 9. CONCLUSION AND WORK REMAINING

This chapter has presented a design for the base language of an extensible language system. It has also presented a technique for semantic specification of programming languages and applied it to the formal definition of the base language. It has demonstrated the utility of applying formal semantic specification in the design of programming languages, rather than to their a posteriori description. Further, it has demonstrated that such a formal semantic specification allows the definition of significant extensions easily, precisely, and clearly. Finally, this chapter has discussed in informal fashion various aspects of the design which were not amenable to formal definition.

However, it should be noted that only a base language has been designed. Considerable study remains to be carried out in designing a core language and embedding this core in a language system. While such study is beyond the scope of this chapter, it seems desirable to delineate the chief topics requiring further attention and sketch out what appear to be the best approaches. One topic is the mechanism for syntax extensions. Another is the system or environment in which the language processor exists. A third is the issue of compilation and its relation to the extension mechanism.

9.1 ADDITIONS TO THE BASE

Before taking up the language core, we wish to point out and deal with a number of omissions in the base. The first of these is the set of primitive data types. As noted in section 3.1, the mode real has not been included in EL1. The existence of floating point hardware strongly suggests

that it should be. This involves the following modifications:

(1) appropriate additions to the concrete syntax so that real constants can be expressed,

(2) addition of a ddb to represent the mode real,

(3) addition of routines to convert between integers and reals,

(4) making the four primitive arithmetic procedures generic, i.e., accepting either reals or ints,

(5) addition of an equality primitive equal-real,

(6) appropriate additions to the construction functions used in mode definition so that new modes can be defined with real components.

None of these presents any major difficulty.

A more fundamental omission is the absence of jumps and labels.† This was an intentional exclusion. It is our contention that, with few exceptions, explicit jumps have a baneful effect on the programs in which they are used. Because they break the relationship between static text and dynamic flow, they make an algorithm more difficult to comprehend or modify. Also, they tend to serve as a substitute for careful analysis in writing the program. That is, sloppy analysis of logical relations usually manifests itself in tortuously complex programs; jumps invite patching over such a maze instead of re-analyzing it properly. E. Dijkstra [Dijk65] describes some experiments he performed comparing Algol 60 programs with rewritten versions of the same algorithms in which jumps were abolished:

---

†It should be pointed out that jumps and labels are logically superfluous. Any program using jumps can be translated into one that does not, using instead iterations and conditionals [Böhm66]. Hence, the issue of language power does not enter into this discussion.

> In all cases tried, however, the program without the goto statements turned out to be shorter and more lucid.

Further, jumps make analysis of algorithms more difficult. As noted by Dijkstra [Dijk68], in a language without jumps, the state of a program can be always characterized by a sequence of textual and loop indices. As these are outside the programmer's control, such index sequences provide independent coordinates in which to describe and analyze the progress of the program.

For our part, we were most concerned with the discipline imposed by the exclusion of jumps and the resulting enhancement in clarity and readability of EL1 programs. Readability is particularly important for EL1 since its semantic specification is expressed as code in the language. In constructing the specification, we found that Dijkstra's experiments were not atypical. Initially, jumps were allowed; their removal consistently made the evaluator constructions, and hence the semantics of the forms they define, more transparent and comprehensible.

However, the exclusion of jumps and labels creates one difficulty. It was argued in section 7.3 that a formal specification should be pragmatically as well as semantically valid. That is, the behavior specified by the formal definition should correspond as closely as possible to that intended for the actual implementation. However, the specification of iteration-form given in section 5.8.4 violates this dictum; it defines iteration by means of a recursive evaluator, whereas it is clear than any reasonable implementation will use a code loop. We could instead give the specification of section 5.8.4 using an iterative evaluator, but this would constitute a genuinely objectionable circularity. It can be well imagined that

with two different interpretations of the semantics of the iteration-form, one could consult the specification and find each confirmed, if the point in question caused two different interpretations of the specification. The best solution seems to be to introduce jumps and labels as a primitive construction. Iteration can then be defined by means of an evaluator which uses a jump; further, a similar evaluator can be used to define jumps. This leaves a circular definition of jumps, but a jump is indeed a more fundamental notion than the admittedly complex iteration-form.

Hence, it appears that our design criteria require the addition of jumps and labels for use in the semantic specification. This raises two sets of questions:

(1) What sorts of scope rules do labels obey? How do labels relate to other data objects – for example, are label-valued variables to be admitted?

(2) What about the above arguments concerning the harmful effects of jumps?

We believe both sets of questions are satisfactorily answered by the following design.

(1) Jumps and labels will be allowed only in the simplest possible form: jumps only within a compound-form. A statement may be labeled by prefixing it with an identifier followed by a colon. Clauses may have as their consequent (cf. §5.7.3) the special format

goto £

where £ is an identifier which labels some statement in the compound-form.

(2) Jumps and labels are provided with the explicit intent that they be used only to define higher-level forms of control. This is not the same

as an exhortation to the programmar to minimize use of gotos. We give a positive injunction: to use jumps in the definition of new forms designed to supplant their appearance.

Another facet of the base which requires addition is input/output. In section 5.15.5, it was assumed that there exist two files – one for input and one for output. To say the least, this is an oversimplification. In view of current equipment and operating systems, it is reasonable for the language to assume that there exists one or more channels (drums, disks, tapes, etc.) each containing one or more files. Channels are identified by logical channel number: 1, 2, 3, ..., max; files are identified by symbolic name. The implementation determines the number of channels, and the correspondence between logical channel numbers and physical channels. Input/output can be either legible (i.e., character form) or nonlegible (e.g., binary). The former is dictated by the representation given for values in the concrete syntax; the latter is implementation-dependent.

A basic set of the file-handling primitives is

```
open_output ~
PROC(channel_number: int, file_name: symbol, file_type: symbol) bool;

open_input ~
PROC(channel_number: int, file_name: symbol) bool;

close ~
PROC(channel_number: int, file_name: symbol) bool;
```

Here, channel-number is an integer between 1 and some implementation-defined maximum, file-name is an arbitrary symbol, and file-type is either "legible" or "nonlegible". The procedures return TRUE if the commanded action was successfully carried out, otherwise FALSE. For

example, if no additional files can be opened on some channel n, then "open_output(n, any_name, "legible")" returns FALSE.

The procedures in section 5.15.5 operate on the standard input and output files. These may be established and changed by two primitives

```
standard_input ~
PROC(channel_number: int, file_name: symbol) none;

standard_output ~
PROC(channel_number: int, file_name: symbol) none;
```

The above procedures taken together with those of section 5.15.5 are sufficient for performing basic input/output. However, it is often desirable to format the data being transmitted, especially on output. Traditionally, programming languages have provided special format constructions which can be used to provide the requisite specifications (e.g., cf. Fortran IV [IBM66c], Cobol [COBOL61], PL/I [IBM66a], and Algol 68 [vanW69]). For Algol 60, where input/output was initially omitted, there have been a number of proposals for adding such format constructions (e.g., [Perlis64] and [Knu64]).

Format constructions have not been provided in EL1, nor is there any need for their addition. One point demonstrated (but not recognized) by the proposals for input/output additions to Algol 60 is that format specification can be obtained entirely as a language extension. It is necessary to provide only a few low-level primitives for transacting with the input/output devices; everything else can be readily built up from these. Because format specification rules tend to be quite complex and idiosyncratic, allowing the programmer control over the format language is particularly desirable.

The required primitives include the 0-ary procedures line and page which skip to the next output line and page if these operations are defined on the standard output file, and position which returns the integer index of the current character position on the standard output file. The atomic output procedure write-char was defined in section 5.15.5; this must be complemented by the symmetric input procedure read-char.

The formatting technique we propose is to convert all values to character string representation and then manipulate the character strings as required. The first step requires a new primitive

```
convert_to_string ~
PROC(x: int_or_bool, i: int BYREF) string;
```

This converts x to a string S, sets i to the number of characters in S, and returns S. The argument x may be either an int or a bool; when reals are added to EL1, the first formal parameter will be changed to admit reals as well. The representation of x by S is that specified by the concrete syntax. Given the string S insertion of dollar signs, check protection signs, special column layout, or other desired text manipulation can be performed using general string manipulation techniques. When the desired string has been created, it is output as a sequence of characters. Formatted input is the inverse: a character string is accepted as input, broken up by string manipulation operations, and the resulting items converted using

```
convert_from_string ~
PROC(s: string; flag: bool BYREF) int_or_bool;
```

This converts s to an int or a bool and returns the result, setting flag to FALSE if conversion cannot be performed.

We anticipate an extension to EL1 for carrying out string manipulation for input/output as well as more general uses, say in the spirit and using the notation of Ambit/S [Chris64], [Chris65]. Using this for formatting I/O has two advantages over a special builtin format language intended for that purpose. (1) It will surely be more general and hence will allow formatting not provided for in the format language. (2) Because it can be used for purposes other than I/O, it will pay its way to an extent not possible for a dedicated format language.

9.2 COMPLETING THE CORE

Judged as a language core, EL1 is found wanting one important characteristic: a syntax extension facility. Most of the mechanism required for this has already been discussed in sections 4.3.1 and 5.1 and will be available in the system. Hence, to provide the facility, we need only give the language appropriate handles on this mechanism.

To allow syntax extension, we require

(1) a means of stating new productions of the concrete syntax and representing productions as data objects of the language,

(2) a parser which will accept new productions,

(3) a means of stating new rules of the abstract syntax,

(4) a means of specifying the mapping from external program representation to internal representation, i.e., from concrete to abstract syntax,

(5) a means of specifying the evaluation rules for a new syntactic construct,

(6) a means of specifying the scope of syntaxes.

The language system already provides for (2), (3) and (5); it is therefore only necessary to address (1), (4), and (6).

To represent productions of the concrete syntax as data objects in EL1, we introduce a new mode, production:

```
element ⇐ S(item: symbol, terminal_flag: bool);
right_part ⇐ R(element);
production ⇐ S(lhs: symbol, rhs: right_part);
```

This allows a program to declare variables of type production, to compute productions, and to change the values of productions by assignment. It will be useful to allow constants of mode production. Hence, we add to the concrete syntax another alternative for constant:

```
constant → bool_constant | int_constant | char_constant |
           noneref_ constant | none_constant | symbol_constant |
           mode_constant | proc_constant | production_constant
```

and define a representation for production-constant so that, for example,

```
expression → expression + term
```

becomes an admissible constant in the reference language. Defining such a representation requires (1) expanding the character set and (2) using an escape character so that the metalinguistic marks "→", "|", "{", "}", "*", "⊛", etc. can be represented. Otherwise, the definition is straightforward.

Since the parser delivers a generation tree, specifying the mapping from concrete to abstract syntax requires that one deal explicitly with generation trees. Hence, we define generation-tree as a new mode. Using this, each syntactic construct is mapped into abstract representation by a procedure supplied as part of the definition of that construct.

There is only one tricky point here: syntactic ambiguity. That is, it may be that a concrete syntax is ambiguous, or becomes ambiguous when

new productions are added to it. Initial experience with extensible languages [Irons70] indicates that this will not be uncommon. Further, it is, in general, recursively undecidable whether a context-free grammar is ambiguous [Cant62] so that we cannot construct an algorithm for detecting such ambiguities on addition of new productions. Hence, it must be assumed that we will, in general, be dealing with an ambiguous concrete syntax. This presents no problem to the parse algorithm, for it tries all paths in parallel. However, it does imply that some provision must be made for handling ambiguity. A sophisticated system might define its generation trees in such fashion that ambiguous generations can be represented and require that the procedures which map from concrete to abstract form be able to accept ambiguous trees and choose one alternative. A simpler, but perhaps more satisfactory, solution is to accept only those programs whose parse is unambiguous. That is, we admit syntaxes with potential ambiguity but deem it a syntax error if such an ambiguity actually occurs in an input string.

Clearly, the second solution is a subcase of the first and can be obtained from it by letting all mappings which are to perform disambiguation instead announce an error. However, the restriction to unambiguous strings is attractive in that it allows each syntax rule to be stated independently of all others. Consider some non-terminal N having k alternatives in the concrete syntax

$$N \rightarrow \alpha_1 | \alpha_2 | \cdots | \alpha_k$$

Corresponding to each alternative, there will in general be a data type for the abstract syntax

$$T_1, T_2, \ldots, T_k$$

If no disambiguation need be carried out then it is necessary only to map instances of α_i onto T_i, for N can be legally construed in only one way. Hence, we can specify a separate mapping from concrete to abstract form for each i, say

$$M_1, M_2, \ldots, M_k$$

The point of this is that when adding a new alternative for N, the new alternative can be stated without reference to or concern with the existing ones. A complete specification for a new alternative is then given by:

(1) a concrete production, $N \rightarrow \alpha_{k+1}$,

(2) an abstract data type, T_{k+1},

(3) a mapping from instances of α_{k+1} to instances of T_{k+1},

(4) an evaluator, E_{k+1}, for the type T_{k+1}.

An example may help to make this technique clear. It will be recalled that the iteration-form of EL1 (cf. §5.8) allows the iterated execution of a form while an index variable steps through a range of values, until some test becomes TRUE. On occasion, it may be useful to specify repetition of a form until some test becomes TRUE, with no need for an index variable. For example,

REPEAT x ←f(x) TILL p(x) > q(x)

specifies that f(x) is to be repeatedly assigned to x, until p(x) is greater than q(x). To simplify the discussion, we adopt the convention that a repetition is to have no useful value, i.e., its value is NOTHING.

Turning to the definition of such repetition forms, we specify that the concrete syntax is

```
form → REPEAT form TILL form
```

while the abstract syntax is

```
repetition ← STRUCT (body: form, test: form)
```

The evaluator for this syntax rule accepts objects of type repetition and obtains their value

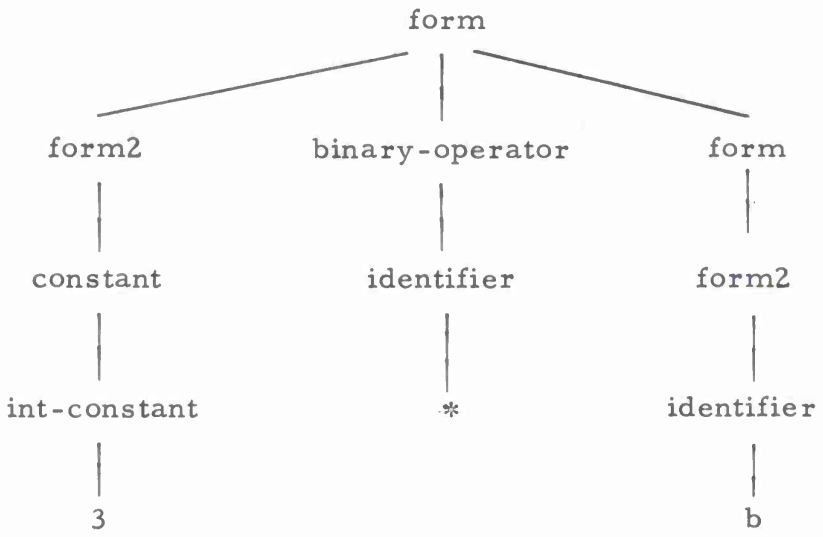
```
ev_repetition ←
PROC(r: repetition) ptr_any;
ev_rep2(r.test, r.body);
NIL ENDP;

ev_rep2 ←
PROC(test: form, body: form) none;
eval_to_type(test, bool) = FALSE ⇒ NOTHING;
eval(body);
ev_rep2(test, body);
ENDP
```

To formalize this syntax rule, it is necessary to specify the mapping from concrete to abstract representation. This in turn requires a representation for generation trees by a data type in the language

```
DECL gen_tree: mode;
gen_tree ← allocate(ddb, ⟨ ⟩);
gen_tree ⇐ S(node: symbol, sons: ROW(PTR(gen_tree)));
```

If g is a gen-tree then g . node is the symbol which heads the tree; g . sons is the set of immediate descendants; g . sons[i] is the i^{th} immediate descendant. A terminal node g has the property that length (g . sons) = 0; i.e., it has no descendants. For example, the generation tree



is represented by the gen-tree g having the property that

g . node = "form",

g . sons[1] . node = "form2",

g . sons[2] . node = "binary_operator",

g . sons[2] . sons[1] . sons[1] . node = "*".

The mapping, i.e., translation, for objects of syntactic type repetition is specified by the following procedure

```
trans_ repetition ←
PROC(g: gen_tree) repetition;
⟨repetition: trans_form(g . sons[2] ), trans_form(g . sons[4] )⟩
ENDP;
```

This constructs an aggregate of mode repetition consisting of two components. The first component is obtained by performing the concrete-to-abstract mapping on the second component of the sub-tree (the first "form" in the concrete syntax rule); the second abstract component is obtained by mapping the fourth concrete component. The mapping of the components is carried out by the procedure trans-form which handles conversion for objects of syntactic type form. Trans-form is defined in an analogous fashion; it returns a form, as required by the definition of aggregate.

To complete the example, it is necessary only to choose some specific notation for syntax rules in the reference language. The following illustrates a plausible choice.

```
SYNTAX_RULE
    form : : = REPEAT form TILL form  &

    repetition ← S(body:form, test:form)  &

    trans_repetition ←
    PROC(g:gen_tree) repetition;
    ⟨repetition:trans_form(g.sons[2]), trans_form(g.sons[4])⟩
    ENDP  &

    ev_repetition ←
    PROC(r:repetition) ptr_any;
    ev_rep2(r.test, r.body); NIL  ENDP,
```

```
ev_rep2 ←
PROC(test:form, body:form) none;
eval_to_type(test, bool) = FALSE ⟹ NOTHING;
eval(body);
ev_rep2(test, body); ENDP
END_RULE
```

This uses the symbols "SYNTAX-RULE" and "END-RULE" as opening and closing brackets, the symbol "&" as a delimiter to separate the four components, and the symbol "::=" to separate the left and right hand sides of the concrete production.

To summarize, a new syntax rule is a 4-tuple:

⟨concrete production, abstract type,
a mapping from concrete to abstract representation, evaluator⟩

A syntax is a set of syntax rules.

Given a syntax S, a new syntax S′ can be defined by adding to it one set of rules S_a and deleting from it another set S_d; that is,

$$S' \leftarrow S \cup S_a - S_d$$

By defining the operators "$\cup$", "-", and "$\leftarrow$" to act on syntaxes, the creation, modification, and extension of syntax sets can be readily carried out.

There is, however, a problem in establishing the syntax scope; i.e., how one specifies what the "current" syntax shall be. In a block-structured language such as Algol 60, it would be convenient to link syntax scope to block scope so that the syntax for each block B_i is

$$S_i \equiv S_{io} \cup S_{ia} - S_{id}$$

where S_{io} is the syntax of the immediately surrounding block and S_{ia} and S_{id} are declared by declarations in the ⟨blockhead⟩ of B_i as syntax sets being added and deleted, respectively. In EL1, however, static block structure does not have the importance it has in Algol 60 so that this scope convention is not really justified for EL1 (cf. §7.2.2). Further, it would be useful to treat syntax sets as manipulatable objects, if only to allow the creation of and subsequent drawing upon a syntax library. It would therefore seem reasonable to make the establishment of a syntax S as the "current" language syntax an executable form.

How this is carried out depends on the environment in which the language exists — specifically the relation between parse-time and run-time. That is, establishing and changing a syntax by executable commands brings up issues in job control and its interaction with program execution. Specifically, one would want different conventions in a batch environment than in an on-line, interactive one. We intend that EL1 operate in an interactive environment; after discussing this environment in the next sub-section, we will discuss how syntax change will be handled.

9.3 INTERACTIVE ENVIRONMENT

EL1 is currently specified as if it were to operate in a batch environment. A program (i.e., form) is read in, it is evaluated, possibly some output is produced, and action halts. While this was a realistic scenario ten years ago and is still the mainstay of computing today, it is hardly appropriate for attacking the interesting problems for which the language can be used. Clearly, EL1 should be augmented to operate in an on-line, interactive environment. We chose the batch processing model only for initial simplicity; here, we discuss the necessary augments.

The construction of an interactive environment is to some degree independent of the language itself. Hence, the environments designed for other on-line languages such as Joss, APL, Lisp and Basic can, in large measure, be taken over for EL1. We contend that the best of these is that used for the BBN Lisp System [Bobr68] and intend to adopt as much of this as possible. We refer the reader to the cited reference for a general discussion of the BBN Lisp System and to [Bobr67b] and [Teit69] for detailed discussion of the particularly important issues: error-handling, editing, and debugging.

The basic change to the EL1 specification is to replace the read-evaluate-halt sequence with a loop[†]

(1) read in a form,

(2) evaluate it,

(3) print its value,

(4) go to (1).

[†] The loop is broken by evaluation of a special procedure, say logout, which returns control to a higher-level supervisor, e.g., the operating system.

The program described by this loop is called the EL1 supervisor.

This cycle would be of little use if each form was evaluated independently of all others, i.e., in a virgin environment. To make it useful, we add the notion of top-level environment: a set of bindings to global variables whose scope is the entire session, rather than a specific procedure. To create such variables, we allow declarations to be executable forms when submitted as programs to the supervisor, so that

```
DECL x:INT
```

is legal at the top-level. Evaluation of this form creates a variable x of mode INT and initializes it to zero. Subsequently, the command[†]

```
x ← 3
```

sets the value of x to be 3. Unlike variables local to a procedure, x may later be redeclared by the command

```
DECL x:complex
```

so that the command

```
x ← ⟨complex:2,10⟩
```

is legal. The scope of global variables such as x is treated as if each command was evaluated within the dynamic scope of a procedure to which all globals are local. That is, any global can be used free in any command. Continuing with the above example, the command

```
PROC(z:complex)none; x.re ← z.im ENDP(⟨complex:5,7⟩)
```

changes the value of x to ⟨complex:7,10⟩.

[†]For emphasis, we will use the term "command" to refer to forms input to the supervisor.

Going one step farther, it will prove convenient to allow use of variables without declaring or redeclaring their mode. For example,

x ← 3

by itself provides quite enough information to both establish a variable named "x" of correct mode and assign the desired value to it. This sort of implied declaration, while hardly profound, is almost a necessity if the system is to provide a comfortable top-level environment.

Implementation of global variables is relatively straightforward. The mode symbol-table-element is redefined to have an additional component global-value of type ptr-any. The routine ev-symbol (cf. §5.5.4) which evaluates symbols is redefined such that if it finds a NIL datum component, it tries the global-value cell. Hence, the global value will have outermost scope and will be used whenever no value of smaller scope supercedes this. The top-level evaluator must be somewhat different from the normal eval (cf. §5.3.5) in that it accepts declarations as legal forms and treats them specially. Specifically, the storage for a global variable is obtained from the heap. Also, to provide for automatic declaration of global variables in situations such as

x ← 3

the top-level evaluator treats assignments having the format

identifier ← form

as a special case. The form is evaluated, producing a value $\mathcal{O}$. If the identifier has no global value or this value is incompatible with the mode of $\mathcal{O}$, then an automatic (re)declaration is performed.

The ability to transact with global values allows the programmer to define procedures by executing a top-level assignment to a proc-var, e.g.,

```
factorial ← PROC(n:INT) INT; n = 0 ⇒ 1; factorial(n+1) ENDP;
```

Editing can be carried out by executing a call on an editing program (which can be written in EL1)

```
edit(factorial)
```

The procedure edit is to act as a lower-level supervisor accepting editing commands and acting upon them, e.g.,

```
replace "+" by "-"
```

This bring up another issue: how editing is to be carried out. Specifically, the question is what structure is to be ascribed to a form in specifying its parts for the purpose of editing them. Taking the form as a single undifferentiated string of characters is the simplest approach but has little else to recommend it. It becomes hopelessly clumsy when dealing with a form of any size. It is far more useful to treat the form as a structured object and specify editing commands using this structure. The natural structure is that assigned by the concrete syntax. Hence, it should be possible to speak of the i^{th} statement of a procedure or the test part of a conditional clause. In short, what we require is syntax-directed editing; i.e., the editor interprets the editing commands in accordance with the syntax for the form under consideration.

Establishing the syntax for a form can be carried out as follows. We introduce a primitive procedure set-current-syntax which when called sets the "current" syntax. Hence, after

```
set_current_syntax(s)
```

the current syntax is s. Whenever a command is input to the supervisor, the parser is called as part of the process of translating the string of input characters to the internal representation of the command. When the parser is called, it uses the current syntax.

Hence, the relation between source text and abstract program is specified by the syntax current at the time the form was parsed. In general, a form is determined by a source text and a syntax. Since the syntax cannot be changed during the reading of a form, the syntax throughout a form is constant. When a command contains many constituent forms, the syntax associated with the outermost form applies throughout. Hence, it might be useful to establish a distinction between a form and a command as follows:

$$\text{command} \Leftarrow S(\text{object}:\text{form},\ \text{interpretation}:\text{syntax})$$

Given a command, its interpretation component allows one to obtain the concrete syntax used in parsing it and, more important, the evaluation rules which specify its meaning.

9.4 COMPILATION

It was argued in section 2.1 that for the purpose of semantic specification, a model based on interpretation was distinctly preferable to one based on compilation. Section 5 presented such a model. Section 6 discussed how the model could be converted to an interpreter to run on a computing machine. Quite aside from issues in formal semantics, this is a reasonable approach to take in obtaining a first implementation. Among other things, an interpreter-based system is generally more suitable for an interactive environment of the sort described in section 9.3. It simplifies program editing and makes practical the construction of programs by programs for immediate execution.† This is particularly useful in debugging: sophisticated tracing and monitoring can be performed at little expense by using the editor to insert conditional breakpoints into procedures.

However, the loose bindings which make possible this flexibility have their price: slow execution. Experience with various Lisp systems [McCar62], [Bobr69] indicates that interpreted code, at least in Lisp, runs one to two orders of magnitude slower than compiled code. Hence, at some point it will be necessary to sacrifice loose bindings and perform compilation. It will prove useful to use the term "compilation" in a somewhat idiosyncratic fashion, to mean the progressive replacement of the variable by the constant. Translation from abstract representation to machine code is only one aspect of this. For obvious reasons, it is useful

---

†While this can surely be done in a compiler-based language by calling on the compiler at run-time, as in Fortran or MAD on the University of Michigan Executive System for the 7090 [Mich64], it is less efficient and convenient and hence not usually done.

to allow various levels of compilation. We distinguish four, of which the first two have already been discussed:

(1) string text (on which character string editing is appropriate),

(2) abstract program representation (edited in syntactic units and evaluated by an interpreter),

(3) function compilation in which a single procedure is translated to machine language (editing within a compiled procedure is forbidden, but individual procedures can be changed in abstract form and recompiled without affecting others),

(4) group compilation in which a group of procedures is compiled as a unit; since the values of all proc-vars in the group are frozen, it is possible to carry out incremental compilation in selecting the appropriate cases from generic procedures in the group (procedures in the group can only be changed by recompiling the entire group).

Integrating a compiler with the existing language requires some changes. Procedures compiled at the function level are already provided for (cf. §5.12.4), since it is assumed that all the primitive procedures are represented in machine-code form. Evaluating arguments and binding these to the formal parameters of code-procedures is roughly the same as for explicit-procedures; the major difference between the two is that the former are executed (cf. §5.13.4). On the other hand, code compiled at the group level is not treated in the current model and extensions must be made to provide for this. The problem is not the abstract syntax definition which is straightforward, but rather that of introducing into the language the notion of "freezing" the values of certain variables – here, the proc-vars of the procedures in the group. There has been some work on this sort of freezing (e.g., cf. [CPL66] and [Pop68]); however, some further

research is needed, particularly with regard to the unification of frozen variables with own variables in the Algol 60 sense.

To allow efficient compilation, it will be necessary to make a few other changes to the language. One such change has been discussed in section 7.1.3: proc-vars should be strongly rather than weakly typed. For example, instead of

```
DECL p: proc_var;
```

it would be useful to provide additional information in the declaration. It would, for example, be nice to know at compile-time the number of arguments to p, their modes, and the result delivered by p independent of any particular procedure which might be assigned to p. For example,

```
DECL p: proc_var(INT, complex, intρ) char;
```

Another piece of information which would be useful to the compiler is the modes of all free variables. A means of providing this information can be integrated nicely into the language if we carry out the extension discussed in section 7.2.1 and allow declared variables to be bound by reference to objects having scope outside the procedure. For example,

```
DECL x: intρ EQU a[i];
```

establishes that within the procedure "x" names the same object as "a[i]" names outside the procedure, where the value of i is fixed at the time of binding. We can then require that all variables used inside the procedure proper be bound; the effect of a free variable x of mode $\mathfrak{M}$ is obtained by

```
DECL x: 𝔐 EQU x;
```

Here, the first "x" is a formal name whose scope is this procedure, and the second "x" refers to the variable which exists outside the procedure.

A paraphrase extension would allow one to write the declaration

FREE x: $\mathfrak{M}$;

by defining this to have the same meaning.

One other aspect of compilation requires considerable study and will likely prove to be an important area for future research: implementation of metaphrase extensions. We introduced the notion of metaphrase extension with the observation that a good semantic formalism describes a space far larger than the particular language it is used to model. Many facilities which are difficult to state as paraphrases on the core language can be easily and efficiently specified by using the formalism to define a metaphrase extension. Section 8, specifically sections 8.3 and 8.4, illustrated how such extensions may be used to define powerful language facilities. This gives a rigorous specification of the extensions; it should be noted, however, that there remains the issue of implementation.

There is, of course, one immediate solution. Let $\mathcal{T}$ be some new syntactic form with evaluator $\mathcal{E}$ specified by an EL1 procedure $\mathcal{P}$. We can simply use $\mathcal{P}$ to evaluate instances $\mathcal{F}$ of $\mathcal{T}$. However, since $\mathcal{P}$ is interpreted and it in turn interprets $\mathcal{F}$, we pay a double overhead; this may be unacceptable. The solution is to take $\mathcal{P}$ as formal specification of $\mathcal{E}$ but transform it into some more efficient means for carrying out the required evaluation. The issue is how to carry this out. Hand translation, either by writing assembly code or by using a translator writing system, is acceptable only as a last resort. It is inelegant and overly demanding on the programmer. Further, it may well introduce errors: if we specify an evaluation process in two different ways, it is unlikely that we have said the same thing twice; very likely we have said two different and contradicting things.

If a compiler is available, it can be used to perform the translation. However, this is subject to two limitations. (1) Let $\mathscr{C}$ be the result of compiling $\mathscr{P}$. $\mathscr{C}$ acts exactly as did $\mathscr{P}$ only faster; i.e., $\mathscr{C}$ still interprets instances of the type $\mathscr{T}$ because $\mathscr{P}$ was written as an interpreter. Hence, although new syntactic types such as $\mathscr{T}$ may be added, instances of these types can be only evaluated by interpretation; there is no compiler for them. The language then has two levels: the original language $\mathfrak{L}_o$ and some extension set $\mathfrak{L}_i$. Since only programs written in strict $\mathfrak{L}_o$ can be compiled, there will be justifiable temptation to stay within its confines, on the ground of efficiency. (2) In particular, $\mathscr{P}$ must be written in $\mathfrak{L}_o$; otherwise, the compiler will reject it. This explicitly rules out the powerful technique of building definition sets one upon another. In short, this sort of approach would significantly weaken the extension facility and its usefulness.

The most promising approach would seem to be the creation of a "smart" compiler for evaluators. That is, given a procedure $\mathscr{P}$ which evaluates by interpretation instances of a syntactic type $\mathscr{T}$, the evaluator-compiler turns out a procedure which can compile instances of $\mathscr{T}$. The evaluator-compiler differs from a related notion, the compiler-compiler, as follows. The source program fed to the latter specifies a compiler in a high-level language; this needs only to be translated into machine code by the compiler-compiler. On the other hand, the source program $\mathscr{P}$ fed to the evaluator-compiler specifies an interpreter; the evaluator-compiler deduces from this an equivalent compiler $\mathscr{P}'$ and then translates this. The difficult point is, of course, performing the deduction. Realizing this proposal will require techniques from compiler construction, artificial intelligence, and the theory of programming languages. We anticipate that this will be a fruitful and instructive area for future research.

9.5 OTHER OPEN ISSUES

In addition to the above issues, there are a number of others which have been neglected in this study. We point these out to make clear their omission and, hopefully, invite research into these areas.

The first is the issue of control. In EL1, control is strictly hierarchal; control can only be relinquished by procedure call and returned by procedure exit. (We informally extended this to provide for interrupts but did not supply details.) However, this is only one of many possibilities; others include: co-routine calls, returns and jumps which skip through one or more levels of intervening procedures, parallel control, and backtracking searches among alternative paths. Some of these could be added to the language by metaphrase extension, for some of the handles on control (e.g., the stack) are available as manipulatable objects in the semantic model. However, such additions would likely be ad hoc mimicking of various control features found in other languages. What is needed is a study into regimes of control, how they relate to one another, possible sets of primitives† which form bases for control structures, and similar issues. Once these issues are understood, incorporation of control structures into an extensible language can be carried out in a systematic, rational fashion.

A second issue is related to control: the sharing of resources among various processes. Here, resources include common files on output

---

†It should be clear that there is no one unique set of control primitives from which all the others can be obtained. There are several such sets. The real issue is working out which primitives are independent of which others. From this, various sets can be tabulated. Selection of a set from such a listing then depends on personal taste, available hardware, and other factors not connected with the formal theory of control.

devices, common portions of address space, and common code segments; processes include both separate tasks initiated by different programs and streams of a given task established by a parallel control facility. We should like such sharing to take place subject to restrictions which provide required protection in various levels. This brings up issues such as ownership of resources, transfer of ownership, and various types of sharing permission. We should like a language cognizant of these facilities and our semantic model expanded to explicate and systematize their behavior. Again, this is an area in need of study – study in which formal semantic models will play a significant role.

REFERENCES

[Abr66] Abrahams, P.W. "The LISP 2 Language and System," AFIPS Proc. Vol. 29 (1966), Fall Joint Computer Conference, pp. 661-676.

[Alb68a] Alber, K. and Oliva, P. Translation of PL/I Into Abstract Syntax, TR 25.086, IBM Laboratory, Vienna, Austria, June 1968.

[Alb68b] Alber, K. et al. Concrete Syntax of PL/I, TR 25.084, IBM Laboratory, Vienna, Austria, June 1968.

[Ard64] Arden, B., Galler, B., and Graham, R. The Michigan Algorithm Decoder, University of Michigan Press, December 1964.

[Bell68] Bell, J.R. The Design of a Minimal Expandable Computer Language, Doctoral Dissertation, Stanford University, December 1968.

[Ben68] Bennett, R. K. BUILD . . . A Basis for Uniform Language Definition, SIG-CR-336, Signatron, Inc., Lexington, Massachusetts, June 1968. (Copies may be also obtained from: Clearinghouse for Federal Scientific and Technical Information.)

[Bobr67a] Bobrow, D.G. and Murphey, D.L. "Structure of a LISP System Using Two-Level Storage," Comm. ACM, Vol. 10, No. 3 (March 1967), pp. 155-159.

[Bobr67b] Bobrow, D.G. and Teitelman, W. Debugging in an On-Line Interactive LISP, Bolt Beranek and Newman, Inc., Cambridge, Massachusetts, November 1967.

[Bobr68] Bobrow, Daniel G. et al., The BBN 940 LISP System, Bolt Beranek and Newman, Inc., Cambridge, Massachusetts, April 1968.

[Bobr69] Bobrow, Daniel G. Personal communication.

[Böhm66] Böhm, Corrado and Jacopini, Guiseppe "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules," Comm. ACM, Vol. 9, No. 5 (May 1966), pp. 366-371.

[Cant62] Cantor, D. "On the Ambiguity Problem of Backus Systems," Journal of the Association for Computing Machinery, No. 9, 1962.

[Chea66] Cheatham, T.E. "The Introduction of Definitional Facilities Into Higher Level Programming Languages," AFIPS Fall Joint Computer Conference, 1966, Vol. 29, pp. 623-637.

[Chea67] Cheatham, T.E., Jr. The Theory and Construction of Compilers, Massachusetts Computer Associates, Inc., Wakefield, Massachusetts, 1967.

[Chris64] Christensen, Carlos AMBIT: A Programming Language for Algebraic Symbol Manipulation, Massachusetts Computer Associates, Inc., Wakefield, Massachusetts, October 1964.

[Chris65] Christensen, Carlos "Examples of Symbol Manipulation in the AMBIT Programming Language," Proc. ACM 20th National Conference, Cleveland, Ohio, August 1965, pp. 247-261.

[Chris66] Christensen, C. and Shaw, C.J. (editors) "Proc. of the Extensible Languages Symposium," in SIGPLAN Notices, Vol. 4, No. 8, August 1969.

[Cobol61] COBOL – 1961, Revised Specifications for a Common Business Oriented Language, May 1961. Copies may be obtained from Superintendent of Documents, U.S. Printing Office, Washington, D.C.

[Coh67] Cohen, Jacques "A Use of Fast and Slow Memories in List-Processing Languages," Comm. ACM, Vol. 10, No. 2 (February 1967), pp. 82-86.

[Coll66] Collins, G.E. "PM, A System for Polynomial Manipulation," Comm. ACM, Vol. 9, No. 8 (August 1966), pp. 578-589.

[CPL66] CPL Working Papers, authored variously by: Buxton, Gray, Park, and Strachey, University of London Institute of Computer Science, 1966.

[Cur58] Curry, H. and Feys, R. Combinatory Logic, Vol. 1, North-Holland Publishing Co., Amsterdam, 1958.

[DeBak67] De Bakker, J.W. Formal Definition of Programming Languages, Mathematisch Centrum, Amsterdam, 1967.

[Deu66] Deuel, P. "On a Storage Mapping Function for Data Structures," Comm. ACM, Vol. 9, No. 5 (May 1966), pp. 344-347.

[Dijk60] Dijkstra, E.W. "Recursive Programming," in Programming Systems and Languages, edited by S. Rosen, McGraw-Hill, New York, 1966.

[Dijk62] Dijkstra, E.W. "An Attempt to Unify the Constituent Concepts of Serial Program Execution," in Symbolic Languages in Data Processing, Gordon and Breach Science Publishers, New York, 1962.

[Dijk65] Dijkstra, Edsger W. "Programming Considered as a Human Activity," in Information Processing 1965, Proceedings of the IFIP Congress 1965, edited by W. A. Kalenich, Spartan Books, Inc., Washington, D.C., 1965.

[Dkjk68] Dijkstra, Edsgar W. "Go To Statements Considered Harmful," letter to the editor, Comm. ACM, Vol. 11, No. 3 (March 1968), pp. 147-148.

[Earl68] Earley, Jay An Efficient Context-Free Parsing Algorithm, doctoral dissertation, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, August 1968.

[Earl69] Earley, J. VERS – An Extendable Language with an Implementation Facility, rough draft, Computer Science Department, University of California of Berkeley, April 1969.

[Evans68] Evans, Arthur "PAL – a Language Designed for Teaching Programming Linguistics," in Proc. 23rd National Conference, Association for Computing Machinery, Brandon/Systems Press, Princeton, New Jersey, 1968.

[Evans69] Evans, A. PAL: Pedagogical Algorithmic Language, Department of Electrical Engineering, M.I.T., Cambridge, Massachusetts, September 1969.

[Feld66] Feldman, J. "A Formal Semantics for Computer Languages and its Application in a Compiler-Compiler," Comm. ACM, Vol. 9, No. 1 (January 1966), pp. 3-9.

[Feld68] Feldman, J. and Gries, D. "Translator Writing Systems," Comm. ACM, Vol. 11, No. 2 (February 1968), pp. 77-113.

[Fen69] Fenichel, R.R. and Yochelson, J.C. "A LISP Garbage Collector for Virtual-Memory Computer Systems," Comm. ACM, Vol. 12, No. 11 (November 1969), pp. 611-612.

[Fle68] Fleck, M. and Neuhold, E. Formal Definition of the PL/I Compile Time Facilities, TR 25.080, IBM Laboratory, Vienna, Austria, June 1968.

[Floy63] Floyd, Robert W. "Syntactic Analysis and Operator Precedence," Journal ACM, Vol. 10 (July 1963), pp. 316-333.

[Gal67] Galler, B.A. and Perlis, A.J. "A Proposal for Definitions in Algol," Comm. ACM, Vol. 10, No. 4 (April 1967), pp. 204-219.

[Gar66] Garwick, J.V. "The Definition of Programming Languages by Their Compilers," in Formal Language Description Languages for Computer Programming, ed. by Steel, T.B., North-Holland Publishing Co., Amsterdam, 1966.

[Gar67] Garwick, J.V. A General Purpose Language (GPL), Intern Rapport S-32, Norwegian Defence Research Establishment, P.O. Box 25, Kjeller, Norway, June 1967.

[Gar68a] Garwick, Jan V. "GPL, a Truly General Purpose Language," Comm. ACM, Vol. 11, No. 9 (September 1968), pp. 634-638.

[Gar68b] Garwick, Jan V. Personal communication at the Working Conference on Extensible Languages, Carnegie-Mellon University, December 1968.

[Ger69] Gerhart, S. A Survey of Extensible Languages, Preliminary draft, The RAND Corporation, Santa Monica, California, August 1969.

[Ger70] Gerhart, S. Formal Definition of APL, Unpublished paper, Computer Science Department, Carnegie Institute of Technology, March 1970.

[Gins66] Ginsburg, S. The Mathematical Theory of Context-Free Languages, McGraw-Hill, New York, 1966.

[Har69] Harrison, M.G., BALM – An Extendable List-Processing Language, NYO-1480-118, Courant Institute of Mathematical Sciences, New York University, June 1969.

[Hoff62] Hoffman, S.A. "Data Structures that Generalize Rectangular Arrays," Proc. AFIPS 1962 Joint Computer Conference, pp. 325-333.

[IBM66a] IBM System/360 Operating System, PL/I Language Specifications, Form C28-6571-3, IBM Systems Library, IBM Corporation, Programming Systems Publications, New York, 1966.

[IBM66b] IBM System/360 Principles of Operation, Form A22-6821-4, IBM Systems Library, IBM Systems Development Division, Poughkeepsie, New York, 1966.

[IBM66c] IBM 7090/7094 IBSYS Operating System, Version 13, Fortran IV Language, Form C28-6390-3, IBM Systems Reference Library, 1966.

[Irons61] Irons, Edgar T. "A Syntax Directed Compiler for ALGOL 60," Comm. ACM, Vol. 4, No. 1 (January 1961), pp. 51-55.

[Irons63] Irons, Edgar T. "An Error-Correcting Parse Algorithm," Comm. ACM, Vol. 6, No. 11 (November 1963), pp. 669-673.

[Irons68] Irons, E.T. IMP, Working Paper No. 229, IDA, Communications Research Division, Princeton, New Jersey, April 1968.

[Irons70] Irons, E.T. "Experience with an Extensible Language," Comm. ACM, Vol. 13, No. 1 (January 1970), pp. 31-40.

[Iver62] Iverson, Kenneth E. A Programming Language, John Wiley and Sons, Inc., New York, 1962.

[John68] Johnson, Walter L. et al. "Automatic Generation of Efficient Lexical Processors Using Finite State Techniques," Comm. ACM, Vol. 11, No. 12 (December 1968), pp. 805-813.

[Jorr69] Jorrand, P. and Hammer, M. M. The Formal Definition of BASEL – Parts 1, 2, and 3, Massachusetts Computer Associates, Inc., Wakefield, Massachusetts, August 1969.

[Kapl68] Kaplan, Donald M. "Some Completeness Results in the Mathematical Theory of Computation," Journal ACM, Vol. 15, No. 1 (January 1968), pp. 124-134.

[Kay68] Kay, A.C. FLEX – A Flexible Extendable Language, Report No. 4-7, Computer Science, University of Utah, June 1968.

[Knu64] Knuth, Donald E. et al. "A Proposal for Input-Output Conventions in ALGOL 60," Comm. ACM, Vol. 7, No. 5 (May 1964), pp. 273-283.

[Knu67] Knuth, Donald E. "The Remaining Trouble Spots in ALGOL 60," Comm. ACM, Vol. 10, No. 10 (October 1967), pp. 611-618.

[Knu68] Knuth, Donald E. The Art of Computer Programming, Vol. 1, Addison-Wesley, Reading, Massachusetts, 1968.

[Land64] Landin, P.J. "The Mechanical Evaluation of Expressions," The Computer Journal, (January 1964), pp. 308-320.

[Land65] Landin, P.J. "A Correspondence Between ALGOL 60 and Church's Lambda-Notation, Parts I and II," Comm. ACM, Vol. 8, Nos. 2 and 3 (February and March 1965), pp. 89-101, 158-165.

[Land66a] Landin, P.J. "A Formal Description of Algol 60," in Formal Language Description Languages, ed. by T. B. Steel, North-Holland Publishing Company, Amsterdam, 1966.

[Land66b] Landin, P.J. "The Next 700 Programming Languages," Comm. ACM, Vol. 9, No. 3 (March 1966), pp. 157-164.

[Land66c] Landin, P.J. "A λ-Calculus Approach," in Advances in Programming and Non-Numerical Computation, ed. by L. Fox, Pergamon Press, New York, 1966.

[Leav66] Leavenworth, B.M. "Syntax Macros and Extended Translation," Comm. ACM, Vol. 9, No. 11 (November 1966), pp. 790-795.

[Lee69] Lee, Jon A.N. "The Vienna Definition Language: A Generalization of Instruction Definitions," SIGPLAN Symposium on Programming Language Definition, San Francisco, California, August 1969.

[Lind69] Lindsey, C.H. and van der Meulen, S.G. Informal Introduction to Algol 68, 4th Draft, working paper of working group WG2.1 IFIP, Printed at: Mathematisch Centrum, Amsterdam, August 1969.

[Luc68a] Lucas, P. et al. Informal Introduction to the Abstract Syntax and Interpretation of PL/I, TR 25.083, IBM Laboratory, Vienna, Austria, June 1968.

[Luc68b] Lucas, P. and Wolk, K. On the Formal Description of PL/I, Report of the IBM Laboratory, Vienna, Austria, December 1968.

[MacLar69] MacLaren, M.D. "Macro Processing in EPS," SIGPLAN Notices, Vol. 4, No. 8 (August 1969), pp. 32-36.

[McCar60] McCarthy, John. "Recursive Functions of Symbolic Expressions and Their Computation by Machine," Comm. ACM, Vol. 3, No. 4 (April 1960), pp. 184-195.

[McCar62] McCarthy, John et al. Lisp 1.5 Programmer's Manual, The M.I.T. Press, Cambridge, Massachusetts, 1962.

[McCar63] McCarthy, John. "Toward a Mathematical Science of Computation," in Information Processing 1962 (Proceedings of the IFIP Congress, 1962) ed. by C. M. Popplewell, North-Holland Publishing Company, Amsterdam, 1963.

[McCar66] McCarthy, John. "A Formal Description of a Subset of Algol," in Formal Language Description Languages, ed. by T. B. Steel, North-Holland Publishing Company, Amsterdam, 1966.

[McCar69] McCarthy, J. Oral communication during panel discussion at: Symposium on Programming Language Definition, sponsored by ACM-SIGPLAN, August 1969.

[McKe66] McKeeman, W.M. An Approach to Computer Language Design, Doctoral dissertation, Stanford University, August 1966.

[Mich64] "Michigan Executive System Subroutines," University of Michigan Executive System for the IBM 7090 Computer, University of Michigan, Ann Arbor, Michigan, September 1964.

[Mills68] Mills, D.L. The Syntactic Structure of MAD/I, CONCOMP Project, The University of Michigan, June 1968.

[Mins69] Minsky, Marvin. Form and Content in Computer Science, Artificial Intelligence Memo No. 187, Massachusetts Institute of Technology, Project MAC, December 1969.

[Mond67] Mondshein, L.F. Vital Compiler-Compiler System Reference Manual, Technical Note 1967-12, M.I.T. Lincoln Laboratory, Lexington, Massachusetts, September 1967.

[Mor68] Morris, James H. Lambda-Calculus Models of Programming Languages, Doctoral dissertation, Massachusetts Institute of Technology, Cambridge, Massachusetts, December 1968.

[Naur60] Naur, P. (ed.) "Report on the Algorithmic Language ALGOL 60," Comm. ACM, Vol. 3, No. 5 (May 1960), pp. 299-314.

[Naur63] Naur, P. (ed.) "Revised Report on the Algorithmic Language ALGOL 60," Comm. ACM, Vol. 6, No. 1 (January 1963), pp. 1-17.

[New68] Newey, M.C. "An Efficient System for User Extendable Languages," AFIPS Proc. Vol. 33 (1968), Fall Joint Computer Conference, pp. 1339-1347.

[Nich68] Nicholls, J.E. Letter by PL/I Language Manager, IBM United Kingdom Laboratories Limited, Development Laboratory, Winchester, Hampshire, England. Letter directed to all recipients of ULD Version 2, dated September 9, 1968.

[Olyn69] Olynyk, F.J. A Proposal for Programmer Defined Variables and Operators, Doctoral dissertation, Department of Engineering, Case Western Reserve University, June 1969.

[Perlis60] Perlis, A.J. "Symbol Manipulation by Threaded Lists," Comm. ACM, Vol. 3, No. 4 (April 1960), pp. 195-204.

[Perlis64] Perlis, A.J. "A Format Language," Comm. ACM, Vol. 7, No. 2 (February 1964), pp. 89-97.

[Perlis66] Perlis, A.J. "The Synthesis of Algorithmic Systems," Proc. of 21st National Conference, Association for Computing Machinery (1966), Thompson Book Company, Washington, D.C., pp. 1-6.

[Pop68] Popplestone, R. J. and Burstall, R.M. "Pop-2 Reference Manual," in Machine Intelligence 2, edited by E. Dale and D. Michie, American Elsevier Publishing Company, Inc., New York, 1968.

[Rand69] Randell, B. "A Note on Storage Fragmentation and Program Segmentation," Comm. ACM, Vol. 12, No. 7 (July 1969), pp. 365-369.

[Rey69] Reynolds, John C. "A Set-Theoretic Approach to the Concept of Type," working paper, NATO Science Committee Conference, Techniques in Software Engineering, Rome, Italy, October 1969.

[Rob65] Roberts, L.G. "Graphical Communication and Control Languages," in Second Congress on the Information Sciences, Spartan Books, Inc., Washington, D.C., 1965.

[Ros66] Rosen, Saul "The Algol Programming Language," in Programming Systems and Languages, edited by S. Rosen, McGraw-Hill, New York, 1966.

[Ross67] Ross, Douglas T. "The AED Free Storage Package," Comm. ACM, Vol. 10, No. 8 (August 1967), pp. 481-492.

[Rovn68] Rovner, Paul D. and Feldman, Jerome A., The Leap Language and Data Structure, M.I.T. Lincoln Laboratory, Lexington, Massachusetts, January 1968.

[Sch67] Schorr, H. and Waite, W.M. "An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures," Comm. ACM, Vol. 10, No. 8 (August 1967), pp. 501-506.

[SDC67] Technical memoranda in the document series TM-3417 for the LISP2 language, System Development Corporation, Santa Monica, California, 1967.

[Stand67] Standish, T.A. A Data Definition Facility for Programming Languages, Carnegie Institute of Technology, Pittsburgh, Pennsylvania, May 1967.

[Stand68] Standish, T.A. A Preliminary Sketch of a Polymorphic Programming Language, Centro de Calculo Electronic, Universidad Nacional de Mexico, June 1968.

[Symb62] Symbolic Languages in Data Processing (Proceedings of the Symposium of the International Computation Center, Rome, March 1962), Gordon and Breach Science Publishers, New York, 1962.

[Teit69] Teitelman, Warren "Toward a Programming Laboratory," NATO Science Committee Conference, Techniques in Software Engineering, Rome, Italy, October 1969.

[vanW62] van Wijngaarden, A. "Generalized Algol," in Symbolic Languages in Data Processing (Proceedings of the Symposium of the International Computation Center, Rome, March 1962), Gordon and Breach Science Publishers, New York, 1962.

[vanW66] van Wijngaarden, A. "Recursive Definition of Syntax and Semantics," in Formal Language Description Languages, edited by T. B. Steel, North-Holland Publishing Company, Amsterdam, 1966.

[vanW69] van Wijngaarden, A. et al. Report on the Algorithmic Language Algol 68, MR 101, Mathematisch Centrum, Amsterdam, February 1969.

[Walk68] Walk, K. et al. Abstract Syntax and Interpretation of PL/I, TR 25.082, IBM Laboratory, Vienna, Austria, June 1968.

[Web67] Weber, H. "A Microprogrammed Implementation of EULER on IBM System/360 Model 30," Comm. ACM, Vol. 10, No. 9 (September 1967), pp. 549-558.

[Wegn69] Wegner, P. Theories of Semantics, Technical Report No. 69-10, Center for Information Sciences, Brown University, September 1969.

[Win69] Winiecki, K.B. and DesMaisons, R.F. THE BRAIN User's Reference Manual, Draft 13, Project TACT, Harvard University, Cambridge, Massachusetts, January 1969.

[Wir66a] Wirth, N. and Weber, H. "EULER: A Generalization of Algol and Its Formal Definition, Part I, Part II," Comm. ACM, Vol. 9, Nos. 1 and 2 (January and February 1966), pp. 13-23 and pp. 89-99.

[Wir66b] Wirth, Niklaus and Hoare, C.A.R. "A Contribution to the Development of ALGOL," Comm. ACM, Vol. 9, No. 6 (June 1966), pp. 413-432.

Unclassified

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)	2a. REPORT SECURITY CLASSIFICATION
Harvard University Cambridge, Massachusetts	UNCLASSIFIED
	2b. GROUP N/A

3. REPORT TITLE

STUDIES IN EXTENSIBLE PROGRAMMING LANGUAGES

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

None

5. AUTHOR(S) (First name, middle initial, last name)

Ben Wegbreit

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
May 1970	435	166

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
F19628-68-C-0101 b. PROJECT NO. 2801 c. TASK: 280102 d.	ESD-TR-70-297 9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)

10. DISTRIBUTION STATEMENT

This document has been approved for public release and sale; its distribution is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
	Directorate of Systems Design & Development, Hq Electronic Systems Division (AFSC), L G Hanscom Field, Bedford, Mass. 01730

13. ABSTRACT

This work is a study of two topics in the development of an extensible programming language, i.e., a high level language with powerful definitional facilities so designed that the language can be extended and thereby tailored for use in a wide variety of computer applications. The first topic is a theoretical treatment of an extension facility for syntax. It generalizes the notion of context-free grammars to allow the syntax of a language to be a function of its generated strings. It studies the formal properties of such grammars and presents an efficient algorithm for parsing their languages. The second topic of this work is a study of the design and formal specification of a base language on which an extensible language system can be built. It employs a formal definition to present a base language, examines the constraints on the design of such language, and discusses how these constraints shape the language. The language includes one extension facility, that for data types; the facility, its design, and its relation to similar facilities in other languages are analyzed.

DD FORM 1 NOV 65 1473

Unclassified

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
programming languages						
extensible programming languages						
base language						
core language						
extension facility						
definition mechanism						
syntax extension						
context-free grammar						
syntax-directed analysis						
extensible context-free grammar						
data type definition						
formal semantic specification						
abstract syntax						
programming language processor						
extension set						
operator extension						