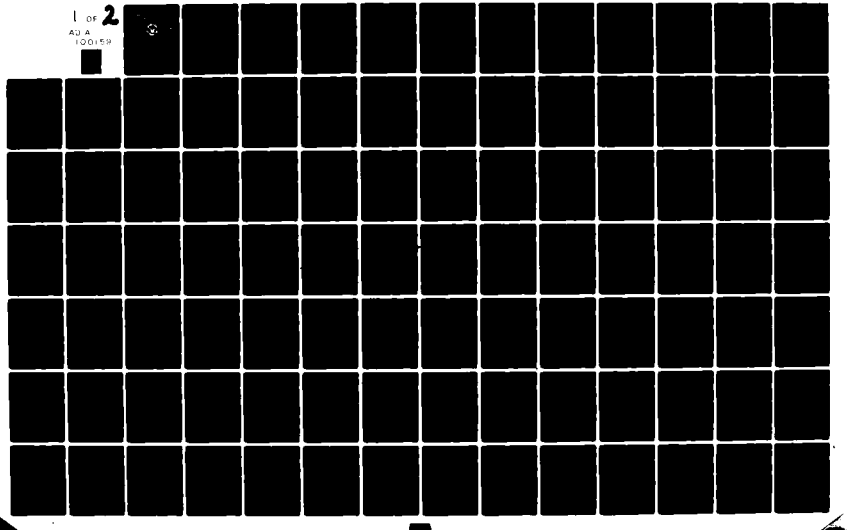
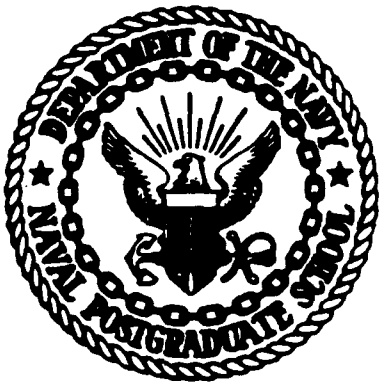
AD-A100 159 NAVAL POSTGRADUATE SCHOOL MONTEREY CA F/G 9/2
A CONCEPTUAL FRAMEWORK FOR GRAMMAR-DRIVEN SYNTHESIS,(U)
DEC 80 W R SHOCKLEY, D P HADDOW

UNCLASSIFIED NL



NAVAL POSTGRADUATE SCHOOL
Monterey, California



THESIS

A CONCEPTUAL FRAMEWORK FOR GRAMMAR-DRIVEN SYNTHESIS

By

William R. Shockley
and
Daniel P. Haddow

December, 1980

Thesis Advisor: Bruce J. MacLennan

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A100159	3. RECIPIENT'S CATALOG NUMBER (9)
4. TITLE (and Subtitle) A Conceptual Framework for Grammar-Driven Synthesis	5. TYPE OF REPORT & PERIOD COVERED Master's Thesis, Dec, 1980
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) William R. Shockley and Daniel P. Haddow	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School, Monterey, CA 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS (1 Dec 80)
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School, Monterey, CA 93940	12. REPORT DATE December 1981
	13. NUMBER OF PAGES 178
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) (12) 179	15. SECURITY CLASS. (of this report) Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release, distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Grammar Driven Synthesis, Grammar Directed Editing, Programming Environments, Syntax Directed Editing

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Conventional parsing techniques use grammars as embedded procedural knowledge bases in mechanisms which are capable of translating words in the language defined into equivalent parse trees.

The approach described in this paper uses context-free grammars as data allowing access to synthesis templates which enable the user to create and interact with parse trees directly.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-014-6601

Unclassified 251450

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

The advantages of this approach are the utility of human-oriented grammars, the dynamic interchangeability of language definitions, immediate error rejection, and the ability to handle partially complete parse trees.

The design for a prototype programming environment using grammar-driven synthesis is presented.

Approved for public release, distribution unlimited

A Conceptual Framework For Grammar-Driven Synthesis

by

William R. Shockley
Lieutenant Commander, United States Navy
B.S., Massachusetts Institute of Technology, 1971

and

Daniel P. Haddow
Lieutenant, United States Navy
B.S., University of Washington, 1974

Submitted in partial fullfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December, 1980

Authors: William R. Shockley

Daniel P. Haddow

Approved by: Bruce MacLennan
Thesis Advisor

Douglas R. Smith
Second Reader

Chairman, Department of Computer Science

D M Woods
Dean of Information and Policy Sciences

ABSTRACT

Conventional parsing techniques use grammars as embedded procedural knowledge bases in mechanisms which are capable of translating words in the language defined into equivalent parse trees. The approach described in this paper uses context-free grammars as data allowing access to synthesis templates which enable the user to create and interact with parse trees directly. The advantages of this approach are the utility of human-oriented grammars, the dynamic interchangeability of language definitions, immediate error rejection, and the ability to handle partially complete parse trees. The design for a prototype programming environment using grammar-driven synthesis is presented.

TABLE OF CONTENTS

I. INTRODUCTION

There is a great deal of interest in the improvement of program and system development efficiency, primarily because software costs have risen dramatically in recent years as a fraction of total system development costs. One approach to the improvement of efficiency is the provision of an enhanced set of interactive program development tools for the programmer and the increased automation of program development. Many such efforts involve the notion of a "programming environment", that is, an interactive environment in which a wide selection of software tools is provided as an integrated package, with a consistent and relatively concise command structure. Typically, a means is provided to allow the programmer to work within the language being used for the program, without having to descend to the object language level to perform any of the functions necessary to create, modify, or test the program.

As a concrete example, the reader's attention is drawn to the most widely-known integrated programming environment, the APL system [Iverson, 1962]. When using this system, the programmer is able to perform all steps in the program development process without ever having to issue explicit commands to the host operating system. The APL environment itself provides an integrated set of facilities for storing, editing, and debugging modules which are arranged in

workspaces and libraries, access to which is available using commands that are part of the APL language definition itself. In addition, so far as the user is concerned, there is no notion of translating, linking, or loading individual functions or programs. To the programmer the system appears to be capable of evaluating programs written in APL without translation, and all of the programmer's interactions with the APL programs defined occur within the syntactic framework of the original source language.

Other language-oriented programming environments are under development or in use, notably the ECL project at Harvard [Wegbreit et. al., 1974], which is based on a LISP-like programming language, and the GANDALF project, [Habermann,1979], which is based on the new Department of Defense language, ADA. Both of these projects are designed to offer an environment which is even more intensively syntax-oriented than that offered by APL. In addition, these systems incorporate into an integrated environment a wide range of facilities normally provided by the host operating system. The two human engineering ideas motivating the design of such systems are to free the programmer from the necessity of learning two command structures, and the ability to reference and access parts of the modules being developed using the natural structure imposed by the syntax of the language in which they are written.

One of the crucial problems which must be solved in implementing such an environment is the need to provide more or less continual access to the evaluable program structure in a syntax-oriented fashion. Conceptually, the system must "understand" the syntactical structure of the program during its entire existence, not simply during the phase in which it is entered into the system. Thus, the internal structure of the program must be sufficiently complex to reflect the syntax of the program at all times, and facilities to utilize this structure must be on-line during the entire period of program development. Since such a requirement must be met for other reasons, a syntax-directed editor is often offered as the primary means of program entry. Such an editor utilizes the on-line knowledge of program structure to allow additions, deletions, and modifications of the program structure to be made based on the natural syntactical units of the program, rather than the more usual line-oriented approach.

Our research was originally motivated by this application for syntax-directed editing, since the program access algorithms for the editor are the very routines involved in program structure access throughout its life in the programming environment. We wished to investigate the task of generating a syntax-directed editor from a grammar description, in the hopes that procedures for routinely

performing such a task could be described in general terms, if not altogether automated. The belief that a set of usable rules could be found was encouraged by the fact that techniques for generating a functionally analogous system, a parser, from a BNF grammar description are well-understood and, in fact, frequently automated.

The techniques reported in this paper are fundamentally very simple, but lie in a direction diametrically opposed to those involved in parser generation. A parser is a mechanism for taking a correct word in some language, and recreating the syntactical structure inherent in that word from the grammar of the language. That this structure can be deduced from what would otherwise be a meaningless string of symbols is a consequence of the fact that the programmer used a grammar to create it that was equivalent to that used by the creator of the parser. The program itself represents a sequentialized version of parallel, hierarchical structures, one in the mind of the programmer, and the other internal to the computer system. The programmer has encoded the structure into the message, and the parser is the mechanism needed to decode it.

Viewed in this light, the use of a parser-based translation system is a very odd solution indeed to the problem of entering a program structure into a computer system for subsequent execution: it is as if a piano were were to be moved it into a house by tearing it into small

pieces, appropriately labelling each one, pushing the pieces through a mail slot, and relying on an automaton inside the house to reassemble the piano. This procedure is notoriously error-prone, and once accomplished, it is extremely difficult for the programmer to gain access in a human-oriented way to the actual structure built. Extending the simile used above, it is as if we could only confirm that the piano had been reconstructed properly by listening to the music emanating from the interior of the house after the piano had been reassembled!

Of course, the historical cause for such a solution is clear: most general-purpose computing systems, at the time language translation technology was elaborated, relied heavily on sequential, batch-oriented input mechanisms such as card readers, and were like houses without front doors, only mail slots. There was a driving need to invent such mechanisms as parsers so that high-level programming could be done at all.

However, with the increased reliance on interactive, remote-entry time-sharing facilities, a radically different solution to the problem of program entry can be investigated. The program structure can be interactively built within the computer in the first place. Such a solution obviates the need for a parser altogether. Instead, the editor and the programmer cooperate to build the desired structure directly. The grammatical

specifications of the language are not used indirectly, to build a decoder for an unnecessary representation, but are used simply as data to guide an appropriate, direct synthesis of a well-structured program representation.

This thesis describes such mechanisms in enough detail to serve as the basis for the implementation of a language independent program entry system. The system is language independent in the sense that data corresponding very closely to the grammar of a context-free language itself, in the form of a finite set of static "transformations", is directly interpreted by the system to form structures well-formed under that grammar. If the grammar data is changed, the same system supports a new language.

We have adopted the term "grammar-driven syntnesis" to describe the function of the systems discussed in this paper, in order to suggest the idea that grammars with a rich set of operators are utilized as knowledge bases with little or no pre-processing. This direct utilization of a human-oriented grammar is to be contrasted, for instance, with the extensive pre-processing required to derive transition tables for driving a shift-reduce parser.

Chapter II describes in very general terms several basic mechanisms for performing such grammar-driven syntnesis, relating them to the fundamental idea of performing a valid derivation under a context-free grammar. Chapter III provides a further elaboration of these mechanisms, aimed

toward the more concrete goal of being able not only to create, but also to modify and delete parts of a hierarchical program structure, in a syntactically consistent way. Chapter IV, which is something of a digression, considers from the viewpoint of database design how programs may be represented and accessed as databases during modification and during storage or transmission from one place or time to another. In Chapter V, a conceptual description is presented of a prototype programming environment, designed to allow the programming language in use to be changed by simply changing the language description installed in the system. This design is concerned solely with the facilities for program modification and entry, and is based on the assumption that a means for describing in a relatively simple way the semantic content of the program structures to be built can be found. Finally, in Chapter VI, the results of the research undertaken so far are summarized, and some suggestions for future investigations are made.

II. GRAMMAR-DRIVEN SYNTHESIS

A. INTRODUCTION

In this chapter, several models for grammar-driven editors of increasing complexity are described in terms of the theory of context-free grammars. Each editor receives two sequences of input symbols, the first representing a context-free grammar, and the second a series of commands which guides the synthesis of a sentential form of the grammar initially provided. The described mechanisms are capable of utilizing very general classes of context-free grammars, including ambiguous and incomplete grammars as well as grammars with useless productions (i.e., productions which do not occur in the derivation sequence for any word of the defined language.) For this reason, we adopt the view that the fundamental product produced by such a synthesizer is a sentential form, possibly containing non-terminal as well as terminal symbols.

The first syntax-directed editor produced by the research group along the lines outlined in this section was written by B. MacLennan in November, 1980 in LISP and called "A Universal Syntax-Directed Editor". The primary motivation for the analysis of grammar-driven synthesis presented in this chapter was to perform an exhaustive review of the algorithms employed and to connect them to the mathematical

theory of context-free grammars in such a way as to justify the adjective "universal", as well as to provide reasonably convincing informal arguments that no critical loopholes had been missed. This technology for using a grammar is compared with conventional parsing techniques, and the feasibility of using such synthesizers as the foundation of a system providing interactive access to a hierarchically organized database (such as that representing an executable program structure) is discussed.

B. GRAMMARS AND SENTENTIAL FORMS

It is assumed that the reader is familiar with the Backus-Naur Form, or BNF, notation for mathematical grammars. Appendix A contains a formal specification for this notational system. The basic concepts from the theory of context-free grammars used throughout this section are adapted from [Hopcroft and Ullman, 1979]. The present section is provided primarily for background and continuity.

A context-free grammar has the following elements:

-- A finite set T of terminal symbols,

-- A finite set N of non-terminal symbols, disjoint from T,

-- A finite set P of productions, each expressed in BNF notation,

-- A designated target non-terminal t included in N.

In addition, for the grammar to be context-free, every production must be of the form

$$< a > ::= X,$$

where X is a string (possibly empty) of terminal and non-terminal symbols, and a is a non-terminal symbol. The acronym "CFG" is commonly used to abbreviate the phrase "context-free grammar". Throughout this chapter, we will adopt the convention of using lower-case letters from the beginning of the alphabet to represent non-terminal symbols, lower-case letters from the end of the alphabet to represent terminal symbols, and upper case letters to represent strings (possibly empty) of terminals and non-terminals. Since we will be considering only context-free grammars, the term "grammar" will always be understood to mean "context-free grammar". We shall also assume that all grammars considered are non-trivial, that is, that the sets T and P are non-empty.

1. Sentential forms.

The basic intuitive concept underlying the idea of a context-free grammar is the notion of derivation: the replacement in a string of a single non-terminal symbol by an equivalent string of terminals and non-terminals as specified by some production.

Let G = (T, N, P, t) be a grammar, and let S(1) and S(2) be strings of symbols. (We adopt the notational convenience of using parenthesized integers to subscript

variable names.) Then we say S(1) derives S(2) in one step, if S(1) and S(2) have the form

$$S(1) = XaZ,\ S(2) = XYZ,$$

and there exists a production in the set P with the form

$$< a > ::= Y.$$

In this case, we write

$$S(1) => S(2).$$

In an analogous fashion, we may define the notion of a leftmost derivation, for which the string X above contains no non-terminal symbols.

A string S is said to derive a string S' in zero or more steps, or simply derive a string S', if one of the following conditions is true: either S = S', or else there exists a series of strings S(1), S(2), . . . , S(n) such that S => S(1), S(1) => S(2), . . ., S(n) => S'. In this case, we write

$$S \ *=> S'.$$

A string W is said to be a sentential form of G if t *=> W, where t is the target symbol of G. A sentential form with no non-terminal symbols is called a word. The set of all such words is called the language defined by G. Such a language is called a context-free language, or "CFL".

A grammar is said to be ambiguous if there exists a word in the language defined by the grammar with two or more distinct leftmost derivations. There exist languages

defined by a context-free grammar that are inherently ambiguous: that is, which cannot be defined by an unambiguous context-free grammar.

2. ARGOT notation.

While BNF notation is convenient for theoretical manipulations because it incorporates a single underlying idea, that of replacement in accordance with a production, a more powerful notation for practical specification of languages is desirable.

For our purposes, we will adapt a system of notation called ARGOT notation, with a concise yet powerful set of replacement operators reminiscent of the operators used in the theory of regular expressions. This notation was developed as the core of a pattern-matching programming language called ARGOT [MacLennan 1975]. In fact, we will use a restricted version of this notation, but it is convenient to introduce the full notation first and then restrict it as required. A formal description of ARGOT notation is provided in Appendix A.

a. Rules and ARGOT expressions.

In place of a set of productions, ARGOT uses a list of named rules, each of the form:

name: expression.

Rule names perform the same role in ARGOT notation as nonterminal symbols in BNF notation; however, it is required that each rule have a unique rule name.

Terminal symbols or strings are denoted by underlinining, use of boldface type, or enclosure by quote marks ("), whichever is appropriate for the typeface available.

The colon corresponds to the BNF metasymbol "::=", separating the rule name from the expression denoting how an occurrence of that rule name may be expanded. Rules are terminated by periods to separate rules unambiguously.

The expression half of a rule is an indefinitely deep hierarchy of elementary replacement operations and sub-expressions, eventually terminating on the deepest levels with terminal strings or rule names. Each operator allows a specific replacement operation, which may be thought of as being applied from the shallowest level of the hierarchy downward in a non-deterministic fashion. Thus, a single ARGOT rule corresponds to a number of equivalent BNF productions.

b. Concatenation

The simplest replacement operator is that of concatenation, or replacement of a single construct by a series of sub-constructs. The concatenation operator is denoted by simple juxtaposition. Concatenated expressions may be grouped into a single construct and used as a sub-expression by means of parentheses. A single BNF production expresses the same idea as a simple ARGOT concatenation

(except that in ARGOT an "empty" rule cannot occur). Thus, the BNF production

<program> ::= program <identifier> <block> .

is equivalent to the ARGOT rule

program: "program" identifier block "." .

The occurrence of a rule name means that that position in the sequence is to be expanded as defined by the named rule, while the occurrence of a terminal string means that that position in the sequence is to be filled by the quoted string.

c. Optional constructs.

An optional sub-expression is surrounded by brackets. The meaning of this operator is that at the specified point, the indicated sub-expression may either be placed into the symbol string or omitted. Thus, the rule

statement: [label] action.

allows replacement of "statement" by either "label action" or by "action".

d. Alternation Operators.

Two alternation operators are provided, simple and optional alternation. Simple alternation is denoted by means of a list of sub-expressions separated by vertical strokes and surrounded by curly brackets. The construct may be expanded by choosing one of the sub-constructs as the replacement. Thus, by the rule

digit: {"0"|"1"|"2"}.

the rule name "digit" may be replaced by any one of "0", "1", or "2".

The optional alternation construct is denoted in the same way as a simple alternation, except that square brackets are used instead of curly brackets. This operator allows replacement not only by any of the indicated alternatives, but also by the empty string. For example, the rule:

sign: ["+" | "-"].

allows the rule name "sign" to be replaced by "+", by "-", or to be deleted (replaced by the empty string).

e. Iteration operators.

Three iteration operators are provided. The required iteration, or simple iteration, is denoted by a plus sign followed by a sub-expression. This construct allows replacement by one or more instances of the sub-expression. Thus, the rule

integer: +digit.

means that an instance of "integer" can be replaced by "digit", by "digit digit", by "digit digit digit", etc.

Optional iteration, denoted by the asterisk followed by a sub-expression, implies that the construct can be replaced by zero or more instances of the sub-expression. Thus, the rule

astring: *"a".

allows expansion of the rule name "astring" to the empty string, or to any of the strings "a", "aa", "aaa", etc.

The final form of iteration, list iteration, is denoted by surrounding two sub-expressions with a sharp sign on the left and three periods on the right. It allows replacement by one or more instances of the first sub-expression, separated by instances of the second sub-expression. Thus, the rule

list: # atom "," ... .

allows replacement of the rule name "list" by "atom", "atom, atom", "atom, atom, atom", etc.

f. Properties of the ARGOT notation.

The most important feature of the notation is, that although it is richer in operators and in this sense more expressive than BNF notation, it is not more powerful. A language is context-free if, and only if, it is expressible as a finite set of ARGOT rules. This can be shown by reducing ARGOT to BNF notation, that is, by providing algorithms for transforming any finite set of context-free BNF productions to an equivalent set of ARGOT rules, and vice-versa. This constructive proof is straightforward and uninformative, as the desired transformations are fairly evident on an intuitive level.

As originally defined, the complete ARGOT programming language, which allows syntactically-keyed computation as well as input and output parameters to be passed between rules, has the full computational power of the

lambda calculus [MacLennan 1975]. The notational subset we are here calling "ARGOT notation" does not have the full power of the ARGOT language defined in this reference.

The notation can also be regarded as a generalization of the notion of a regular expression. We may think of a set of ARGOT rules as being a set of named regular expressions, and then allow rules to refer to themselves directly or indirectly to achieve the power of a context-free grammar. This notational similarity allows the simple statement of a sufficient (but not necessary) condition for the regularity of an ARGOT-defined language. If a finite set of ARGOT rules can be arranged in such an order that the right-hand side of each rule refers only to rules occurring further down the list, the language defined is regular. That this is so can be seen fairly readily. Such an ordering allows replacement of each rule name except for that of the target by the right-hand side of each of the named rules in a terminating sequence. The resulting single rule is simply a regular expression with operators and terminal strings alone on the right-hand side.

This result is of practical use, since if we know that a language is regular, then we know that simple (non-recursive) algorithms exist for processing it. The algorithms for processing it are considerably less complicated than if the language is context-free but not regular, in which case some sort of recursive mechanism is required.

3. Restricted ARGOT notation (R-ARGOT).

The full ARGOT notation, as described, has more expressive power than required for the application we are interested in, for two reasons:

-- its indefinitely nested structure requires recursive routines to access the sub-expressions in a rule, and

-- highly nested expressions are too complicated to express easily-learned syntax units for the user.

That the notation allows indefinite nesting is implied by the fact that the notation itself is an inherently context-free language. Since we shall be accessing the grammatical descriptions of languages as databases, it is highly desirable to be able to describe and encode simple, efficient access routines. In addition, a simpler notation will allow us to conceptualize a given grammar as consisting of a collection of rules each of which is formatted in one of a finite number of ways.

What we would like is a notation that is expressible as a regular expression (as is BNF notation) so that it is easily processed, but retains an adequate amount of expressive power. These goals are met by appropriately restricting the nesting allowed within ARGOT expressions. The resulting notation is called R-ARGOT notation (for either restricted or regular ARGOT).

The set of available operators is restricted to concatenation, required iteration, simple alternation, list

iteration, and the optional operator. The other operators are rendered superfluous by the nesting restriction.

R-ARGOT expressions (rule right-hand sides) may be simple or complex. A simple expression is a concatenation of one or more terminal strings, rule names, or optional rule names. A complex expression is an alternation, required iteration, or list iteration. Any sub-expression in an alternation or iteration must be a rule-name. The first sub-expression in a list operation must be a rule-name. The second may be either a rule-name or terminal string.

The effect of these rules is to limit the number of possible formats available for the grammar designer to a small set. Alternations and simple iteration operators will always be the topmost operator in a given rule expression if they occur at all, and the operands will be simple rule-names in such expressions. The list iteration operator must also be topmost, and only the second operand may be other than a rule-name, and if so, must be a single terminal string. Only if the concatenation operator is topmost may the operands be alternations, and even in this case no further operators are allowed in the rule.

It is something of a surprise that such stringent restrictions result in grammars that are reasonably well-oriented toward human comprehension. The rules that result, when they are read informally, seem to express natural

syntactic units. It must be admitted that an improvement in human comprehensibility might be attained by allowing one level of nesting. However, the simplifications in the rule-access algorithms provided by naming each sub-expression are so striking we have been led to retain R-ARGOT as described here.

The languages defined in Appendices A and B are defined using the R-ARGOT notation. In particular, the reader's attention is drawn to Appendix B, which contains a grammar for the PASCAL programming language. Most of the syntactic rules can be seen to correspond to natural syntactic constructs within the language in a way that BNF productions do not.

One irritation encountered in the use of R-ARGOT is the implicit requirement to rename terminal strings which carry semantic information (that is, that occur as alternatives within an alternation). Where we would like to write, for instance, rules such as

```
string: + character.
character: ( "a" ! "b" ! . . . ! "z" ).
```

we must instead write

```
string: + character.
character: ( a | b | . . . | z ).
a: "a".
b: "b".
. . .
z: "z".
```

To avoid the necessity to provide a large number of trivial rules renaming tokens, we shall assume the existence of a facility in the system for escaping from the normal mode of grammar-driven synthesis to predefined lexical synthesizers. Such a facility is analogous to the separation of the analysis task between the parser and scanner in a conventional compiler. Thus, we will assume that predefined rules exist with such names as "identifier", "integer", "string", etc. In the system to be implemented, these rule names correspond to predefined input scanners and parsers available to the language implementer.

C. A SIMPLE GRAMMAR-DRIVEN STRING EDITOR

In this section, a simple mechanism is described capable of generating sentential forms from an input grammar in BNF notation. This mechanism serves as the fundamental model for grammar-driven editing using interactive production selection to direct the course of the synthesis.

1. The Basic Mechanism.

We may think of the basic mechanism, which will be hereafter referred to as a Grammar-Driven String Editor (GDSE), as a multitape Turing Machine with two input tapes, labeled PHASE1 INPUT and PHASE2 INPUT, four internal tapes labeled GRAMMAR, BUFFER, CURSOR, and PRODUCTION, and an output tape labeled OUTPUT. The PHASE1 INPUT tape contains a context-free BNF grammar, which is stored internally on the GRAMMAR tape. The PHASE2 INPUT tape contains a series of editing commands which will be more fully described shortly. The BUFFER tape is used as a work area to synthesize a sentential form. The CURSOR and PRODUCTION tapes are used to hold indefinitely large integers which number the nonterminal in the BUFFER currently being expanded, and the production being applied from the GRAMMAR tape, respectively. The OUTPUT tape is provided simply as a conceptual convenience: it is used to model the transfer of the final form produced to secondary storage.

The operation of the mechanism is as follows:

a. Phase One -- Copy and Check Grammar.

The PHASE1 INPUT tape is copied onto the GRAMMAR tape. As this is done, the contents of the input tape are parsed in accordance with the grammar listed in Appendix A for BNF notation. Since this grammar is regular, the input tape can be rejected or accepted as a legitimate context-free grammar in a finite number of steps. Without loss of

generality, we assume that the first production names the target symbol as its left-hand side.

b. Phase Two -- Initialization.

In phase two, the mechanism is used to generate sentential forms via valid derivation steps on the BUFFER tape. First, the target non-terminal is copied from the first production onto the BUFFER tape. Then the following loop is executed. Each cycle corresponds to one step of a valid derivation.

c. Phase Two -- Loop.

A symbol is read from the PHASE2 INPUT tape. If it is 'Q' (for 'Quit'), control is passed to the next step beyond the loop.

If the order to quit is not received, two integers are copied from the PHASE2 INPUT tape. These integers are assumed to encode the relative position in the buffer of the next non-terminal to be replaced, and the production in the grammar to be used to replace it. Both of the integers must be checked to be sure that they refer to a real non-terminal in the BUFFER and to a real production in the GRAMMAR. If they do, the left-hand side of the selected production is checked to make sure it is the same as the selected non-terminal. If any of these checks fail, the integers are simply ignored and the loop re-entered from the beginning. Otherwise, the indicated replacement is performed. In detail, the mechanism performs the following

steps.

First, an integer (suitably encoded) is read from PHASE2 INPUT and placed in the CURSOR register. Suppose this integer is N. The N'th non-terminal symbol on the BUFFER tape is located. If there is none, control is returned to the top of the loop.

Another integer is then read from PHASE2 INPUT and copied onto the PRODUCTION tape. Suppose it is M. The M'th production is located: if there is none, control is returned to the top of the loop.

The heads are then moved to the N'th non-terminal on the BUFFER tape, and the left-hand side of the M'th production, and the two non-terminals compared. If they are not the same, control is returned to the top of the loop.

If they are the same, the right-hand side of the M'th production is used to replace the N'th non-terminal on the BUFFER tape, moving characters to the right to make room for the new symbols as needed.

Finally, control is returned to the top of the loop.

d. Phase 2 -- End.

The BUFFER tape is copied to OUTPUT and the machine halts, accepting.

e. Synopsis.

The algorithm described is nothing more than a restatement, in somewhat more detailed terms, of the fundamental method for producing some valid sentential form under a context-free grammar. Determinism has been introduced by using an additional input phase, which encodes, as the derivation proceeds, choices for the next non-terminal to be expanded and the production to be used. Erroneous input during this phase is ignored. This simple mechanism captures the essential flavor of grammar-driven synthesis. We may note that the contents of the PHASE2 INPUT tape may be obtained in sequence when they are needed, and are never re-used. Thus, this input process serves as an entirely adequate model for an interactive process. Throughout the remainder of this section, we will assume that the "Phase Two User" is able to examine the internal state of the machine in order to determine the current state of the synthesis and decide what to do next. We make this assumption to avoid cluttering the mechanism descriptions with output routines, which do not have any impact on the current state of the synthesis in any event.

2. Properties of the GDSE.

The fundamental property possessed by the GDSE is that it never contains an invalid form in the BUFFER, and that a PHASE2 INPUT string exists which will cause the machine to halt, accepting, with any desired sentential form

on the OUTPUT tape.

In one sense, these assertions are hardly susceptible to a convincing proof, since the mechanism is so obviously related to the notion of valid derivation in the first place that any proof is likely to be less convincing than this intuition. The proof can be carried through based on an induction over the number of times the mechanism passes through the loop. Since the BUFFER contains a valid sentential form (the target symbol) when the loop is entered the first time, and each step in the loop either leaves the BUFFER unchanged or changes one valid form to another by expanding a single non-terminal in accordance with a production in the input grammar, the BUFFER contains a valid sentential form whenever the loop is entered. When the 'Q' symbol is read, the last form generated is placed on the OUTPUT tape prior to acceptance. (The machine may reject if the 'Q' symbol is missing).

Given a desired sentential form, there exists some valid derivation sequence, starting with the target symbol, such that each derives in one step the next, and the last is the desired form. (There may be more than one such sequence of steps). Each step consists of selection of a non-terminal in the last derivation, and its replacement by the right-hand side of some production. Thus, given the list of derivation steps, it is easy to construct a list of pairs of integers for the PHASE2 INPUT tape which will recreate these

steps in the BUFFER. Hence for any sentential form, there exists a PHASE2 INPUT tape which will cause that form to appear in the BUFFER. Appending a 'Q' on this tape will cause the machine to halt, accepting, with the desired form on the OUTPUT tape.

3. Discussion.

As previously mentioned, although conceptually simple, the GDSE is the underlying model for all of our more elaborate grammar-driven mechanisms. The GDSE plays a role for grammar-driven synthesizers analogous to that played by a Deterministic Push-Down Automaton (DPDA) for parser-based systems. The fundamental simplicity of grammar-driven synthesizers arises from the fact that this underlying mechanism is a direct restatement, with determinism incorporated, of the very notion of a sequence of steps in a valid derivation. The resulting simplicity is to be contrasted with the much more complicated "set of items" construction required to generate the DPDA associated with a grammar, which causes the relation between a grammar and its parser to be very indirect [Aho and Ullman 1977]. The GDSE utilizes the grammar directly to synthesize words, rather than using it indirectly to produce a derivative mechanism able to decode words.

We might note that we have allowed the output of the GDSE to be any valid sentential form, not requiring it to be composed of strictly terminal symbols. In other words, we

are taking as the fundamental entity defined by a grammar, a sentential form instead of a word. It is easy enough to fix up the mechanism so that before halting, it checks the string in the BUFFER for non-terminals and accepts only if there are none. Our decision not to do so is based on the philosophy that additional restrictions should not be introduced so long as the output without them is sensible. In practical terms, a valid sentential form under a grammar for a programming language corresponds to a partially complete, yet well-structured program, with the missing parts labeled appropriately by non-terminal symbols. In fact, the ability to deal with such "reasonable" partial programs is one of the primary advantages of a programming system based on grammar-driven synthesis.

Retaining this capability yields an even more interesting property. No problem develops if the GDSE encounters a non-terminal in the right-hand side of some production which is undefined. Once this non-terminal is copied into the BUFFER it can never be replaced, so once this action has been taken a word will never be derived. However, the use of an undefined non-terminal can yield a class of sentential forms. In the context of grammars defining programming languages, the described situation might occur if some subset of the complete grammar for the target language was in use. The resulting form would be meaningful, and lead to a complete program, once the complete

grammar were defined.

Thus, we see that the class of grammar-driven synthesizers to be described have the ability to deal intelligently not only with partial programs, but also with partially-complete grammars, in a natural way.

Finally, we note that ambiguous grammars present no problem for the GDSE. If the input grammar is ambiguous, this simply means that there is more than one way to generate at least one sentential form.

The question that remains to be answered is whether grammar-driven synthesizers can be used to synthesize more interesting constructs than strings (for instance, some data structure encoding the algorithm represented by the word.). In addition, it is desirable to use a more human-oriented input code. In the remainder of this chapter, first the command, and then the synthesis capabilities will be improved. The resulting mechanisms will inherit the basic properties of the GDSE, however, which remains our fundamental model for grammar-driven synthesis.

D. AN IMPROVED GRAMMAR-DRIVEN STRING EDITOR

In this section we improve the Phase Two command mechanism for the GDSE. The R-ARGOT notation is our primary tool for doing this. This notation provides for a concise and human-oriented set of rules as the grammar definition, allows automatic expansion of rule names when there is only

one way for expansion to be done, and provides a framework for selection of alternative expansion paths based on keying the desired alternative by means of a mnemonic keystroke. Yet the regularity of the notation allows synthesis to proceed in a straight-forward, non-recursive fashion, primarily because the contents of the rule can be accessed by a finite automaton. These properties are not coincidental, since the desire to achieve them provided the primary motivation for restricting the ARGOT notation in the way chosen.

1. Rules and transformations.

We eventually would like to classify every possible rule name replacement according to some finitely-expressible scheme. To this end, we distinguish between the terms "rule" and "transformation". For BNF notation, each production can result in one, and only one, transformation of a non-terminal symbol to a string of symbols. For ARGOT and R-ARGOT notation, in contrast, each rule may express more than one such permissible transformation. The limited nesting of R-ARGOT operators allows us to list all of the transformations allowed for an R-ARGOT grammar in a finite list.

In order to further reduce the set of transformations possible, we introduce a special class of symbols which are assumed to be distinct from either rule names or terminal strings, which we will call "e-symbols". They have

the purpose of serving as place markers in a sentential form, indicating points where optional strings formed according to a particular transformation may be inserted. We will use three classes of such symbols, with the notation "o(rule name)", "i(rule name)", and "l(rule name)". The characters "o", "i" and "l" will be used to encode the exact sort of transformation by which the symbol can be replaced, and the rule name argument will allow the mechanism to access the symbols in the grammar by which they can be replaced. Since their expansion is optional, for output purposes we may think of all of these symbols as representing the empty string. When the buffer is to be copied to output, these symbols are simply skipped.

With this notation in hand, we examine the four sorts of R-ARGOT rules: concatenations, alternations, iterations, and list iterations.

Concatenations involve replacement of the rule name by a sequence of terminal symbols, rule names, and optional rule names. These elements must occur in order exactly as specified in the rule. Any optional rule names are converted to the e-symbol "o(rule name)" when they are encountered. Thus, the rule:

array-type: [packed] "array" "[" ranges "]" "of" type.

allows replacement of the rule name <array> in the buffer by

o(packed) array [<ranges>] of <type>

(In this section, we shall delimit rule names in the buffer

with angle brackets so that they cannot be confused with terminal strings.) If the symbol "o(packed)" is never replaced, this string would be copied to the output tape simply as

array [<ranges>] of <type>

We see that a concatenation rule explicitly stands for a single, invariant transformation. Implicit in the existence of an optional field, however, is an additional transformation of the form

o(rule name) => <rule name>

The use of an e-symbol has allowed us to express what would have been one transformation with an indefinite format, as an indefinitely long (but finite) list of transformations, each of fixed format. This notational trick will be further used in the next chapter to make the list of transformations associated with a grammar even more regular.

Alternation rules are always of the form:

name: { name1 ¦ name2 ¦ . . . ¦ name-n }

and correspond to n transformations:

<name> => <name1>

<name> => <name2>

. . .

<name> => <name-n>

Iteration rules correspond to two transformations: that performed when the rule name is first replaced, and that corresponding to additional iterations. Thus, a rule of the

form:

name: + name1

corresponds to the two transformations:

<name> => <name1> i(name)

i(name) => <name1> i(name)

List iteration rules similarly consist of two transformations. A rule of the form:

name: # name1 name2 ...

corresponds to the transformations:

<name> => <name1> l(name)

l(name)=> <name2> <name1> l(name)

2. Automatic synthesis.

Having listed all possible transformations, we may now determine which of them can be performed automatically. Given a rule name, the type of rule is effectively computable from the form of the right-hand side of the rule alone. If the rule is an alternation, the user must be consulted in order to determine which of the n possible transformations is required. If the rule is a concatenation, there is only one possible expansion. If the rule is a simple iteration or list iteration, the initial transformation is required and should be automatically performed. It may be recalled that predefined rule names (such as "identifier") are allowed in an R-ARGOT grammar to symbolize calls to predefined input scanners. Such rule names do not admit to expan-

sion by rule, but must be expanded by referral to the predefined scanner which may solicit data from the user. Hence, predefined rules cannot be automatically expanded. There is one other possibility: the rule name may be undefined. In this case, no expansion of any kind is possible.

Terminal symbols, by definition, cannot be expanded. The e-symbols all require user attention so also cannot be automatically expanded.

As a matter of terminology, we may classify symbols in the buffer as bound, free, or transient.

Bound symbols are those which admit to no further replacement. Thus, in our system undefined rule names and terminal symbols are bound.

Free symbols are those which require a decision as to whether or not they are to be replaced at all, or by what transformation they are to be replaced. The free symbols are thus names for alternation rules and predefined rules, as well as the e-symbols.

The remaining symbols can be transformed by one, and only one, transformation which is not optional. They represent intermediate steps of a required replacement sequence, may be automatically replaced without restricting the range of words which can be formed from the sentential form currently in the buffer, and thus may be regarded as "transient" in the sense that they are retained only until they are recognized and replaced by their equivalent

automatically. The transient symbols in the described system are names of concatenations, iterations, and list iterations.

Since the expansion of transient symbols can only be done in one way, at the beginning of each Phase Two loop we would like to search the buffer for a transient symbol and expand each one found, continuing this process until there all symbols are either free or bound. Unfortunately, for unrestricted R-ARGOT grammars, there is no guarantee that this process will terminate. If one can start with a concatenation, iteration, or list iteration rule and reach the same rule by applying a sequence of rules not including any optional or alternation rule, the described process may never terminate. Therefore, we must restrict the grammar so that no such cycles exist.

Fortunately, the existence or non-existence of such cycles can be effectively computed given an otherwise syntactically correct R-ARGOT grammar. This restriction is the only semantic constraint we place on R-ARGOT grammars for the remainder of the discussion. The loss in expressive power is not great. Such cycles correspond to recursive expressions with no trivial case in BNF-described languages, and once entered, derive only forms with non-terminals and never words.

With this restriction, which can be enforced by checking the input grammar during Phase One, we now may

allow automatic expansion of transient symbols during the beginning of the Phase Two loop prior to any further processing with the understanding that such expansion is to be performed until no transient symbols remain. With the grammar restricted as described, this process must always terminate. Since the grammar is context-free, the order in which transient symbols are expanded is of no consequence. We will refer to the automatic expansion of all transient symbols until none remain as "autoscanning" .

The addition of the autoscanning feature relieves the Phase Two user of the burden of having to order expansions that are required by the grammar. The price paid for this facility is that only those forms can be produced which consist entirely of bound and free symbols. In the context of a programming language defined by a grammar, the system will now synthesize as much of the program as is syntactically deducible from the part of the program already created by the user.

As a concrete example, we display the results of autoscanning the target symbol for the PASCAL grammar listed in Appendix B:

```
program <identifier> ( <identifier> l<filelist> ) ;
    o(labels)
    o(constants)
    o(types)
    o(variables)
    o(subroutines)
    begin
        <statement>
        l(statements)
    end.
```

3. Improved Cursor Control.

The next improvement to be described is a more useful method of cursor placement.

From the analysis above, we see that after autoscanning is performed, the buffer will contain only bound and free symbols. By definition, the only symbols requiring Phase Two input data for further expansion are free symbols, since bound symbols admit to no expansion at all. It follows that the cursor should always rest on a free symbol. If there are no free symbols, there are no symbols left to expand in the buffer, and the loop may be left, the buffer copied to the output tape, and the algorithm terminated. In general, however, one or more free symbols will be left in the buffer at the end of autoscan. We wish to allow the user a means to move the cursor between them, and must also decide what to do after the symbol indicated by the cursor has been expanded. It should be clear that cursor movement never has any effect on either the contents of the buffer nor on the valid derivations reachable at any point in the synthesis. The first is true simply because cursor movement

leaves the buffer unchanged, and the second because of the context-free nature of the expansion operation.

Accordingly, after autoscanning, if there are any free symbols left, we allow the user to move the cursor back and forth by entering zero or more cursor control symbols (represented by "->" for movement right and by "<-" for movement left).

The only question remaining is how to position the cursor initially, and how to reposition it after a symbol is expanded. We assume that after a symbol is expanded, the buffer is autoscanned again to remove any new transient symbols. If the section of the buffer replacing the expanded symbol now contains one or more free symbols, the cursor is placed at the leftmost such symbol. Otherwise, it is placed at the first free symbol in the remaining string of symbols. If there are none, wraparound takes place and the cursor is placed at the first free symbol in the old substring to the left. Initially, the cursor is placed at the first free symbol in the buffer.

4. Transformation Selection.

Finally, we address the problem of causing an optional transformation to be applied, once the cursor has been positioned as desired by the user.

From the discussions above, the cursor must be resting on a free symbol, that is, at either a predefined rule name or the rule name for an alternation, or at an e-symbol

of type o, i or l. To simplify the command language model, the entry of a blank is adopted as the uniform means of indicating that an expansion is to take place at the current cursor position. If the cursor is at a predefined rule name, control is then turned over to the indicated predefined input scanner. If it is at an e-symbol, the appropriate transformation is made, the result autoscanned, and the cursor repositioned for another loop through the cycle. Finally, if the cursor is at the rule name for an alternation, one of many potential transformations must be selected. Another symbol is entered and this is matched to keystrokes included in the rule body.

Thus, we must extend the R-ARGOT notation to allow inclusion of the keystroke for each alternative which will trigger it. An alternation now looks like:

```
statement: { 'a' assignment
           ! 'i' if-statement
           ! 'w' while-statement
           ! 'c' case-statement
           }.
```

The symbol 'a' will invoke the transformation

<statement> => <assignment>

the symbol 'w' the transformation

<statement> => <while-statement>

and so on.

Extensions to this simple system are easy to implement and desirable. In particular, a string of more than one character could be allowed as key. Some work has been done in allowing a "fall-through" key, symbolized by " '' ", which invokes the indicated transition upon any symbol which does not occur anywhere else in the list of alternative keys, and reapplies the entered symbol to the next alternative generated. Such enhancements are not considered further in the present work.

Thus, the only data which must be entered during Phase Two are cursor control commands, which leave the synthesized string intact but move the cursor, and invocations of transformations, which consist of a single blank, followed by nothing for e-symbol expansions (lists, iterations, or optional field inclusion), by a context- dependent keystroke for alternative selection, and by whatever is needed by the appropriate input scanner for such items as identifiers, numbers, and the like.

5. Discussion.

We have now enhanced the capabilities of the GDSE on the input side to allow string synthesis driven by a human-oriented grammar, with a reasonably supple means of cursor control and transformation selection. The resulting mechanism still has the desirable properties of the GDSE: it can accept virtually any context-free grammar (we have lost those which contain irreducible recursions) and generate any

form derivable under that grammar (some of which are automatically expanded). It is also still true that the buffer never contains an incorrect sentential form.

The mechanism that has been described in this section is considerably simpler than that for a parser generator. This simplicity is the result of allowing interaction between the user and the synthesizer during the stage when the grammar of the language is available to the mechanism. User-provided data is available to guide a true top-down synthesis of the desired word in the defined language.

The described system is highly useful in its own right. It could be used, for instance, to prepare programs for entry into a conventional system with the guarantee that the program was syntactically correct. The compiler used would not need the ability to handle syntactic errors (a notably difficult design problem). In addition, since the input grammar is interpreted, the same editor could be used for many different languages.

We want to do more, however. In the next section, we investigate one way to synthesize more complicated data structures using the grammar-driven editor we have described in this section.

E. TREE SYNTHESIS

So far, all of the mechanisms described synthesize strings. In order to subsume the ideas already developed

under the general notion of tree synthesis, we first characterize strings as a special sort of tree. We then discuss the notion of parse trees, and generalize it to form the more general class of derivation trees, of which both string trees and parse trees are a special case. Since trees are a well-understood data structure, we shall not define them formally but treat their general properties in an intuitive fashion. For the remainder of this section we shall assume that the algorithms necessary to create and manipulate generalized (multi-children), ordered trees are freely available. Such trees consist of a finite number of nodes, each of which has a finite number of children occuring in an ordered sequence.

In addition to having children, we assume that each node may also contain an *indefinite amount* of symbolic information. In particular, with each node may be associated a string called its label.

Those nodes of a tree with no children are its leaf nodes. Since the tree is ordered, its leaf nodes may also be ordered into a linear list. We assume that all of the nodes of a synthesized tree may be examined and accessed for the information they may contain.

1. Re-Interpretation of the GDSE.

In all of the work that follows, we use a synthesizer that is formally identical to the GDSE. We shall call such a mechanism a GDE, for Grammar-Driven Editor. The

action taken by those steps in the algorithm that actually interact with the BUFFER are re-interpreted as calls to tree-manipulation subroutines. The BUFFER is now conceived to contain, not strings of symbols, but appropriately implemented ordered trees with labeled nodes. Rather than describing the algorithms involved to create, modify, and traverse such structures in detail, we assume that mathematically correct subroutines are available to perform the needed functions, since methods for implementing trees using a sequentially-addressed, rewritable memory store are well-known.

In order to re-interpret the improved GDSE as a tree synthesizer in this way, we need routines to initialize the BUFFER with a target tree (or initial tree), move the cursor back and forth, and replace a "symbol" with a "string of symbols" (whatever these terms mean in the new context). Also, we now need to explicitly identify the precise means used to "display" a tree.

Supposing that appropriate routines are available, we wish to argue that the new mechanism, which synthesizes trees, instead of strings, inherits all of the formal properties of the original, in the following sense.

The display algorithm in use may be thought of as a function, d, mapping trees into strings. We shall consider a tree to be a "sentential form" of the input grammar of interest if, and only if, its image is a string which is a

sentential form of the grammar.

We wish to compare the operation of the old and the new mechanisms, given exactly the same stream of input symbols on the PHASE2 INPUT tape, supposing that the grammar specifications on the PHASE1 INPUT tape are equivalent in some as yet unspecified sense. The fundamental property that gives the GDSE all of the features that make it an appropriate synthesizer for sentential forms is that at each entry to the loop, the BUFFER always contains a correct form. This property is a consequence of the fact that the manipulations inside the loop either leave the contents of the buffer unchanged, or transform one valid form to another. Since the BUFFER is initialized with a valid form, by induction the BUFFER never contains anything but a valid form upon loop entry.

We would like the new mechanism to perform the same derivation steps, given the same PHASE2 input sequence, as the old. The display function would then serve as a morphism from the new mechanism to the old, over the operations defined by the possible BUFFER transactions made available by the algorithm within its basic loop. Thus, if it is true that, for any given cycle through the loop by the parallel mechanisms, with identical forms in the two BUFFERS at the beginning of the loop (as viewed under the display function for the new mechanism), and that corresponding derivations are undertaken within the loop, then for every possible

derivation sequence that can occur under the old mechanism there will be one, and only one, derivation sequence which occurs under the new mechanism, and the product of the new mechanism, when viewed under the display function, will be identical to that of the old.

The question of paramount interest, is under what circumstances will this property, that the contents of both BUFFERs will be display-equivalent for any step in equivalent machines, be true?

It is well outside of the scope of our research to provide a complete answer to this question, in the form of a set of necessary and sufficient constraints so that the desired property (which we might call "stepwise equivalence") is true. Rather, we shall provide a description in general terms of a natural class of re-interpretation constraints that are merely sufficient.

In the improved GDSE, the PHASE1 INPUT tape contained a finite set of rules, each of which consisted of a finite set of transformations with one symbol on the left-hand side, and a string of symbols on the right-hand side. In the re-interpreted synthesizer, each transformation will consist of a specification calling for the replacement of a single leaf node, labelled with the symbol on the left-hand side of the original transformation, with a forest of adjacent siblings with leaf nodes labelled with each of the symbols on the right-hand side. Such a tree transformation

specification will be referred to as a template. "Replacement of a symbol by a string" is now taken to mean the replacement of a labelled leaf node by the forest of adjacent siblings specified by the appropriate template.

In order to ensure that the structure in the BUFFER is always a tree, (since we may allow replacement of a node by a forest), it is necessary to ensure that the root node in the BUFFER is never broken up into a forest. We therefore impose the constraint on the system that the BUFFER be initialized with a tree consisting of a special root node with one child, labeled with the target symbol. Since only leaf nodes are ever replaced, no replacement ever turns a previously internal node into a leaf node (no transformations have empty right-hand sides). Since the root node is initially internal, it is never replaced. Hence the structure in the BUFFER is always a bona fide tree.

The above suppositions are insufficient to obtain the stepwise equivalence property by themselves, since we have not addressed the display function, which is used to define what is meant by a tree which is a valid sentential form.

In the final system to be described, the language implementer will be given the power both to select a particular template from all of the valid candidate templates available, corresponding to the given transformation, and also influence the display order of the children of a given

node. The retention of stepwise equivalence depends jointly on the consistent application of this facility, and it is our present intention to provide a sufficient condition which does, in fact, preserve it.

Selection of a single template for each transformation in the original grammar may be thought of as specifying a function, mapping transformations into templates. Let us name this function f.

In the work immediately following, the display algorithm will be very simple. A tree is displayed by listing the labels for all of its leaf nodes in order. Since the right-hand side of templates are ordered forests, we may also speak consistently of applying d to the template: again, we simply list all of the leaf node labels in order. The required constraint is simply this: f and d must be inverse functions on the set of transformations in the grammar and selected templates. That is, each template must display as the transformation to which it corresponds. Finally, movement of the cursor back and forth is to be interpreted as movement of the cursor from leaf node to leaf node, as ordered under the display function.

Under these conditions, stepwise equivalence will be retained by the new mechanism. The fundamental reason for this is that the display algorithm defined is, itself, "context-free". If a given tree is a sentential form, application of a template to it will yield a tree which is

also a sentential form. Moreover, the new tree will display as the same form as that yielded by the corresponding symbol replacement applied by the string synthesizer. Cursor movement also takes place in parallel.

Since the new mechanism is stepwise equivalent to the old, it inherits all of the formal properties of the old. Of course, since the actual contents of the BUFFER may be substantially richer in structure at any given time, the new mechanism may have emergent properties of its own in addition to those inherited from the GDSE, but such properties can be utilized only by using an additional algorithm to access information that has been hidden in internal nodes of the tree in the BUFFER.

A more flexible display algorithm will be used in the final system. The implementer will have the power to permute the display order of the nodes in a template, as well as to display strings stored with the rule instead of as labels of a node. The display algorithm retains the basic property of providing a context-free display, however, and the same constraint applies to the display and template specifications chosen: each template must, in fact, display as its corresponding transformation in order for the system to maintain stepwise equivalence.

2. Strings as Trees.

We may think of a string as a special sort of tree which has a root node and one child for each symbol in the string. Such a two-level tree we shall call a string tree. For instance, the string

"if <expression> then <statement> o(else-part)"

corresponds to the string tree

<root>

if <expression> then <statement> o(else-part)

In order to synthesize string trees with a GDE, we initialize the BUFFER with the tree

<root>

<target>

Replacement of a symbol by a string of symbols is redefined as the replacement of a leaf node by a set of adjacent sibling nodes, fitted into the place of the replaced node in the ordered list of leaf nodes. In other words, the template corresponding to a given transformation is just an ordered forest of single-node trees.

The resulting GDE, although it does synthesize trees, constitutes a system that is isomorphic to the GDSE.

3. Parse Trees.

The concept of a parse tree occurs frequently in the theory of context-free grammars.

We can view parse trees as the structures synthesized by another re-interpretation of the basic grammar-driven synthesizer. The initial tree is taken to be the same, two node tree as for the case of string trees. The notion of replacement of a symbol by a string is reinterpreted as the addition of children to a leaf node, labeled with all the symbols of the string. In other words, templates always take the form of a tree, with the root node labeled with the left-hand side of the transformation, and each child labeled with the appropriate symbol from the right-hand side. As usual, the "string" in the BUFFER is the ordered list of leaf nodes. The resulting structure is considerably richer than that retained in the BUFFER by the GDSE, since once a node is created, it is never removed. (More accurately, if it is removed while a leaf node, it is immediately replaced by a copy of itself.).

4. Comparison of String Trees and Parse Trees.

We take the view that string trees and parse trees are two special cases of a whole range of trees that can represent a particular sentential form. This observation can be justified by comparing the properties of the two types of trees. A string tree incorporates the minimum amount of historical information concerning the derivation sequence by which it was produced: just enough for further derivation to correctly proceed. As a result, string trees are very compact.

Parse trees, on the other hand, incorporate a very large amount of information concerning the derivation sequence by which they were produced: enough so that the entire sequence can be reconstructed (down to the permutation of commutative non-terminal selection). As a result, parse trees are very large. As a concrete example, Figure 1 in Appendix H contains both the parse tree for a trivial PASCAL program.

Our eventual goal is to provide for grammar-driven synthesis of directly evaluable trees of reasonable size. A secondary goal is to do this in such a way that the resulting tree can be displayed as a program in the language in which it was created, but can be evaluated without any additional syntactical access.

Neither string trees nor parse trees are suitable constructs for achieving these goals. String trees incorporated no structural information and must be reparsed in order to access their semantic contents in the correct order. (This process may even be impossible if the string tree was synthesized under an ambiguous grammar.) Too much information has been discarded at the time of synthesis.

On the other hand, parse trees are unreasonably large. Most of the nodes record syntactical information that is semantically content-free.

Our task, therefore, is to find a way to reach some middle ground, synthesizing trees which contain enough nodes to retain the desired control structure, but allowing the elimination of nodes which have no semantic content.

The purpose of the present section is not to provide a complete description of how this is to be done, but to provide a conceptual range of intermediate possibilities. It will then be possible to choose the sort of tree to be synthesized to meet a particular requirement intelligently. In short, we wish to introduce some "engineering slack" into the formal system.

This purpose is realized by introducing the notion of derivation trees; a general concept of which both parse and string trees are a special case.

5. Derivation Trees.

One way to characterize the structure of a parse tree is to note that every parent node in the tree derives its children in exactly one step. Thus, the relation between parents and children in the tree is the same as the "=>" relationship.

We consider the set of trees in which each parent derives its children in zero or more steps; that is, incorporates the "*=>" relationship.

Such trees may be constructed from a parse tree in the following manner:

a. Mark the root and leaf nodes.

b. Mark zero or more of the remaining nodes.

c. Discard each unmarked node. Every time a node is discarded, replace it within the set of its siblings by all of its children, taken now as adjacent siblings. (This procedure preserves the relative ancestry of all undiscarded nodes.).

The above procedure assures that every remaining node derives its new children in zero or more steps. This can be seen by noting that the hypothesis is true for the original parse tree, and that if true for a discarded node and its children, is true for the node's parents and its children during each application of the third step. Hence, it is true for the resulting tree.

In the procedure just specified, the selection of interior nodes to be retained is done non-deterministically. It is the specification of the particular agorithm to be used for selecting nodes for retention that we make available to the system implementer as an engineering choice. The two simplest algorithms are to retain all interior nodes, in which case parse trees are produced, or to discard all interior nodes, in which case string trees are produced.

The trees produced by the procedure just described we call generalized derivation trees. Our goal, however, is not to produce a full parse tree and only then to prune it,

but to synthesize a pruned derivation tree directly as we go along.

This desire suggests that we apply a particular synthesis uniformally, in the sense that for each transformation implicit in the R-ARGOT grammar there be associated one, and only one, synthesis action. This suggestion is not quite a necessary implication: one could conceive of some history or context-dependent algorithm for selecting one of several predefined synthesis actions associated with a transformation. In fact, such "intelligent" systems are an interesting subject for future research.

But if the simpler protocol is adopted, we obtain a sub-class of derivation trees, which we call derivation trees constructed by rule. Both parse trees and string trees are also members of this class. Hereafter, the term "derivation tree" will be understood in this restricted sense.

The association of one, and only one template, with each transformation is very clearly an embodiment of this idea. The GDE previously described is thus a mechanism capable of synthesizing any class of uniform derivation trees desired for a given grammar in R-ARGOT.

In essence, the next chapter represents the selection of further constraints on the template formats to be associated with each type of transformation, in such a way that our design goals are acheived. The trees produced

under the set of protocols are a particular sort of derivation tree constructed by rule, which we shall call hereafter abstract syntax trees. This name is adopted from the ideas contained in [McKeenan 1970] as representing an intermediate stage in the translation of some program in which a parse tree has had its syntax-dependent, semantically void interior nodes pruned away.

b. Elimination of Terminal Strings in Derivation Trees.

An inspection of parse trees such as the one displayed in Figure 1 suggests three general classes of nodes for elimination: those representing a series of production steps needed to fill a high-level slot with a low-level construct (so-called "empty productions"); those encoding options available but not so far taken (e-symbols); and those representing keywords and punctuation.

As the next chapter shows, selection of appropriate template protocols allows removal of nodes representing empty productions. It is our belief that nodes of the second type can also be eliminated by appropriate template selection and context-sensitive computation to compute the existence of a "virtual" option.

We now investigate a methodology for eliminating most nodes required to hold terminal strings.

We first make the observation that most such nodes are semantically content-free. An examination of the K-ARGOT notation will show that terminal symbols can only be

added to a synthesis in one of two ways: by means of a concatenation or list-iteration transformation, or by means of a predefined (autoparsed) rule name expansion. In the second case, the included string may well be meaningful, e.g. if it is an identifier or the like. In the former case, however, since the required terminal string cannot be an optional field, there is no choice as to whether the string can or cannot be included. If such a choice existed, it must have been via an earlier option or alternative selection, and by the template protocols specified in the next chapter, this selection is already encoded into the structure of the tree. There is thus no reason to add a node to the tree simply to represent an invariant field.

On the other hand, in order to be usable we must be able to display the string as if it were a node in the tree. The solution to this quandary is to make provision for computing the location and contents of such virtual fields when the need arises. This can be done, provided that list and concatenation rule templates always have a single head node which can be associated with the specific rule from which they were derived in some way (either by inserting a reference to the rule into the node, or computing the rule from context). If the contents of the virtual fields associated with the rule are then stored with the rule, we can avoid repeating these strings throughout the derivation tree.

These ideas are more concretely discussed in the protocols for template construction in the next chapter.

F. COMPARISON OF GRAMMAR-UTILIZATION TECHNOLOGIES

It is appropriate at this point to step back and place the system of grammar utilization described in this chapter within the range of currently available technologies for grammar utilization. We shall compare this system with the two common parsing techniques: bottom-up and top-down parsing. All three of these techniques may be thought of as producing as output derivation trees.

It should be recognized that the tree produced by a parser in contemporary translation systems is usually "virtual". The parser emits a series of syntax-directed action commands which may be thought of as the sequential representation of a post-order traversal of a derivation tree. The "back end" of the system may be thought of as traversing behind the parser, destroying nodes as quickly as they are built.

Both of the parsing techniques are designed to proceed automatically, that is, without any human intervention. The grammar-driven synthesizer, in comparison, is inherently interactive. This property is both an advantage and a disadvantage, in that the synthesizer utilizes interaction to attain desirable goals, but cannot be implemented without interactive devices being available.

The need for the parser-oriented techniques to proceed automatically places a set of mathematical constraints on the grammars usable by such systems. The grammar-driven synthesizer is capable of utilizing almost any context-free grammar; a capability that allows the language designer to optimize the grammar selected for realizing some programming language towards a set of semantically natural rules which will be easy for the human user to understand.

The parser-based systems are essentially decoders, translating a valid word in the defined language into a more complicated, but equivalent, structure. Inherent in this process is the requirement for the user to use some other system, such as a keypunch or text editor, to formulate a valid input word in sequential form: a notoriously error-prone and tedious process. In contrast, the grammar-driven synthesizer allows the user to create the desired tree structure directly and with no possibility of syntactic error (since such errors are simply rejected immediately).

Finally, we note that both parsing techniques synthesize the output tree from the bottom up. The grammar-driven synthesizer follows a true top-down synthesis: thus, the partially-complete structure is completely well-structured so far as it goes. The system is for this reason well-suited as a base for dealing with partially complete programs.

III. CONCEPTUAL DESIGN FOR GDE

A. INTRODUCTION

In this chapter a conceptual design for a Grammar Directed Editor is developed within the framework defined in Chapter II.

The mathematical model provides a large framework in which to design a Grammar Directed Editor, subject to the following restrictions:

1. Grammar rules are limited to the concatenation, alternation, iteration, list, predefined, and undefined rules in the forms specified by the R-ARGOT notation.

2. The templates associated with these grammar rules may consist of arbitrary forests of siblings, the leaves of which must be labelled in accordance with the transformations summarized in Figure 2.

3. The templates for list and concatenation rules which include terminal symbols must create head nodes which retain or refer to those terminal symbols for display.

A Grammar Directed Editor constructed in accordance with these restrictions will produce a derivation tree whose leaves and terminal symbols, retained in head nodes, are displayable as a valid derivation of the input grammar.

The following design restrictions and goals serve as a basis for limiting the very general nature of the possible

templates to a set of generic templates which define the permissible transformations available for the construction of an Abstract Syntax Tree (AST):

1. The AST should contain the minimum number of nodes consistent with the retention of all necessary semantic and schematic information.

2. The structure of the AST should admit efficient editing algorithms, in particular for append, delete, and insert functions.

3. The AST should not only be an evaluable structure, but further it should require no "preprocessing" between editing and evaluation operations.

4. The generic transformation template structure should be such that the creation of specific templates for a given grammar can be automated over the simplest possible input data, perhaps as simple as a grammar in a suitable notation.

The methodology employed in the design process described in the following section is to apply, working within the constraints which the mathematical model suggests, such further constraints and definitions as may be necessary to develop generic templates for each transformation which realize the design goals. In section C, a method for displaying the AST is developed which is consistent with the generic templates as well as with the requirement that the valid derivation which the AST represents be displayable as such. Section D introduces the notion of a Language

Definition, wherein an R-ARGOT grammar is translated into an ordered collection of transformation templates and display schemas which serves as the basis for the construction and display of an AST.

B. TRANSFORMATIONS

1. Operators and Rulenames

Figure 2 is the result of precisely defining the leaves produced by each of the transformations defined in Chapter II.

A simple change in notation produces Figure 3, wherein every rulename in a transformation is associated with an operator to form a two-part label, as follows:

<r> = NT,r

copt(r) = COPT,r

iopt(r) = IOPT,r

lopt(r) = LOPT,r

pdf(p) = PDF(p),p

where r is any grammar rulename and p is any predefined rulename. The first part of a label, the operator, will guide future transformations. The second part, the rulename, serves as a reference to that section of the language-specific data base containing the information required for performing transformations or display. In other words, labels may be thought of as a self-modifying "program" for the Grammar Directed Editor stored in the

hierarchical AST structure by previous versions of the program, encoding all of the information necessary for subsequent modifications or display of the structure.

Note that as a result of the notational convention adopted here that the set of possible labels is finite over a finite set of grammar rules and, therefore, the set of templates required for such a grammar is also finite. Further, the type of transformation which may be applied to a given node is determined entirely by the operator and rule type association stored within that node.

The alternation and predefined transformations present a problem, however: although the "NT" opcode is usually stored in transient nodes, these two particular transformations must be stored in free nodes. The alternation requires that the user select one of the possible alternatives, and the predefined functions require that the user input a string which they then process. This irregularity is resolved by the introduction of two new operators ALT and TERM and the following pairs of transformations:

```
NT,a    =>  ALT,a
ALT,a   =>  { NT,r1 | ... | NT,rn }
NT,p    =>  TERM,p
TERM,p  =>  PDF(p),p
```

The operators "ALT" and "TERM" may be thought of as logically equivalent to "NT", but as explicitly labelling (for display purposes) the nodes as free (for synthesis

purposes). Figure 4 reflects these modifications to the general transformation table.

The introduction of the two new labels ALT,a and TERM,p, while not altering the leaves produced by the original transformations and thus not violating the validity of the mathematical model's results to systems based on this extension, provide the following benefits:

a. The format for the five defined types of template sets is more regular. At least two transformations are associated with each rule type. The first of these transformations is, in every case, a required transformation. The second and following transformations require some form of interaction with the user.

b. Every node whose label has an "NT" operator may be automatically expanded during the autoscan process. Thus, after autoscan, the only leaves whose labels contain the "NT" operator will be those corresponding to undefined rules.

c. Since for every unique label there is one and only one transformation possible, no contextual information need be extracted from the AST in order to select and perform the correct transformation. This simplifies the tasks both of language implementation as well as AST formation since production and invokation of a transformation template is independent of any AST contextual considerations.

2. Transformation Restrictions

The transformations as discussed so far define only the leaves of a possible forest of siblings which are to replace a particular node of the AST. We now turn our attention to designing the interior structure, if any, of the forests generated by the transformation templates. In the absence of other design goals or restrictions, the driving motivation in determining the forest structure is to obtain as much simplicity and economy of space as possible. These goals must be balanced with the necessity to retain semantic or schematic information to preserve the valid derivation property, as well as to retain sufficient structural information so that insertion and deletion editing functions may be convenient for the user as well as efficient algorithmically. The requirement to be able to delete synthesized subtrees turns out to constrain the template structures such that the other goals are also met.

In order to recover gracefully from erroneously constructed portions of the AST, the user should have the capability to delete any node in the AST, which, as for any hierarchical structure, inevitably involves the ability to delete any subtree. The valid derivation property of the AST requires that deletion of a subtree from an AST be realized as the replacement of the entire subtree by a node which can validly derive that subtree and which also forms a valid derivation with the remainder of the AST. The choice

of the transformation to be applied to a node in the AST is based solely on the information contained in the node itself and is completely independent of the node's context. Therefore, deletion of a subtree must be equivalent to replacement of that subtree by a node with the same label, that is, the same operator and rulename, which the node which was expanded to form the deleted subtree contained when the node was originally created. The constraints provided by the abstract model of Chapter II are not sufficient to guarantee that this can be consistently and efficiently accomplished. For example, consider a grammar which has only concatenation rules, each of which is entirely either nonterminal symbols or terminal symbols. Since the model allows the definition of templates for concatenation rules which have no terminal symbols without a head node, the tree derived from such a grammar could be a string tree, containing no information for reconstructing a node being considered for deletion. The only action possible for a deletion algorithm in this case would be to delete the entire tree. However, consider the effect of the following proposed restrictions:

a. All immediate children of a (necessarily bound) node must be created by the transformations of the rule by which their father was bound.

b. When a node is bound, the rule whose transformation bound the node is permanently recorded in the node.

c. A given transformation may generate two or more childless siblings, or a subtree of the current node, but not both.

d. If a subtree is created by a transformation, it is limited to at most a single generation of children and may consist of a single node.

Given these restrictions, the rule (and therefore, at worst, a choice between two transformation templates) which originally created any given node in the AST can be identified by examining its father. Computation on the father rule templates allows retrieval of the unique node from which the subtree to be deleted was formed. This uniqueness is further discussed below.

3. Transformation Templates

Given the restrictions developed in the previous section, we are prepared to define the forests produced by each of the eleven transformations. The notation utilized in the transformation templates below is defined in Appendix C.

a. Concatenation

Rule:

c : x1 x2 ... xn , xk = { rk ! "["rk"]" ! tk }

Template:

```
                headop,c ( # { NT,rk    if xk = rk
                             | COPT,rk if xk = "["rk"]" }
                             ";" ... ) if for some k,
NT,c =>                                        xk = { rk | "["rk"]" }
                headop,c                 if for all k, xk in T
                       headop = { HEAD | predefined function }
```

There are six cases to be considered in the transformation to be applied to the label NT,c:

	nonterminals	terminals	comment
Case 1:	0	NO	undefined rule
Case 2:	1	NO	useless production
Case 3:	>1	NO	head required by delete
Case 4:	0	YES	terminals only
Case 5:	1	YES	head required by model
Case 6:	>1	YES	head required by model

Case 1 corresponds to the undefined rule wherein no righthand side of the rule exists. The undefined rule transformation is discussed below.

In cases 3, 5, and 6 it is required that a head node be created, in cases 5 and 6 by the mathematical model for the retention of terminal information and in all cases by the restrictions defined for the deletion algorithms. In each case the head node replaces the nonterminal under transformation and the nonterminal and/or optional children

are realized as the immediate children of the head node.

In case 4 a head node retaining the terminal information replaces the nonterminal being transformed. Since there are no nonterminals in the grammar rule for which this form of this transformation is utilized, no children are created. Note that this node is bound since it is transformed into a node which is not one of the label forms for which transformations are defined; in fact, this is the only bound leaf node form generated outside the realm of predefined functions.

Case 2 is the useless production. We could, without violating any of the restrictions thus far imposed, define this case of this transformation as a single node replacement, i.e., as NT,c => NT,r, thus avoiding the creation of a head node carrying no information. However, we see the useless production as a very rare and usually unnecessary occurrence which does not justify the increased algorithmic complexity required for its detection. Therefore, it is treated in the same manner as cases 3, 5, and 6.
Implicit Template:

COPT,r => NT,r

This label must be accompanied by some form of user attention in order that the transformation be invoked, the nature of which is discussed in the next section. Assuming for the moment that the user has elected to take

the option, the transformation applied is a single node replacement wherein the operator COPT is overwritten with NT, and the rulename remains unchanged.

Note that the rulename in the COPT label may be any of the six rule types, including undefined, which raises the question of where to store the template for this transformation. The solution is to make this transformation implicit, that is, to apply the transformation without an explicit template being stored in the grammatical data base. This may be done since the transformation is invariant over all rules in any grammar, depending only on the requisite user attention and the COPT operator.

b. Alternation

Rule:

a : "(" r1 "¦" r2 "¦" ... "¦" rn ")"

Template 1:

NT,a => ALT,a

The transformation for the label NT,a is a single node replacement; the operator NT is replaced with ALT, and the rulename remains unchanged.

Template 2:

```
                NT,rk    if user input valid
ALT,a    =>
                ALT,a    otherwise
```

This label must be accompanied by user input indicating which of the alternatives is desired; suppose for the moment it is the kth. The transformation applied is a

single node replacement wherein the operator ALT becomes NT and the alternation rulename is overwritten with the rulename of the kth alternative. If the user input does not correspond to any of the alternatives, the transformation returns the node unchanged.

c. Iteration

Rule:

```
i : "+" r
```

Template 1:

```
NT, i   =>  ITER,i (  NT,r ; IOPT,i )
```

While not required by the mathematical model, a head node is created by the transformation for the label NT,i to fulfill the deletion requirements. The two leaves specified by the model are formed as the immediate children of the head node in which the operator NT was replaced by ITER. A side effect of the invariant creation of a head node is that, while inconsistent with the model, terminal information applicable to every real child in the iteration sibling string, as opposed to the trailing IOPT child, could be included in the iteration rule if an appropriate extension were made to the R-ARGOT notation.

Template 2:

```
IOPT,i  =>  NT,r ; IOPT,i
```

Triggered by the appropriate user input, the transformation for the label IOPT,i replaces the node with a

pair of siblings which are the leaves required by the model. Note that the rulename in the IOPT label is the same rulename which bound its father. Thus, all children of the ITER node, whether formed when the ITER node was bound or subsequently when the IOPT node was expanded, are formed by one of the transformations under the rulename stored in the ITER node, as required.

d. List

Rule:

1 : "#" r1 x "..." , x = (r2 | "["r2"]" | t)

Template 1:

NT,1 => LIST,1 (NT,r1 ; LOPT,1)

The transformation for the label NT,1 replaces the operator NT with the operator LIST, forming a head node as required by the model in the case the second right-hand-side argument of the grammar rule is a nonterminal and in every case by the deletion requirements. The required leaves form a sibling string under the LIST node.

Template 2:

		NT,r2 ; NT,r1 ; LOPT,1	if x = r2
LOPT,1	=>	COPT,r2 ; NT,r1 ; LOPT,	if x = "["r2"]"
		NT,r1 ; LOPT,1	if x = t

The transformation for this label has three forms, as indicated, for the three possible cases. In all cases, the LOPT node being transformed is replaced with a

sibling string as shown, the nodes of which are the required leaves. As in the IOPT transformation, the LOPT label carries the same rulename as its father so that all children created under a LIST head node are derived from a common parent rule.

e. Predefined

Rule:

p : pdf

Template 1:

NT,p => TERM,p

The transformation for the label NT,p is a single node replacement, the NT operator being overwritten with TERM and the rulename remaining unchanged.

Template 2:

TERM,p => PDF(p,string),p if PDF(p,string) valid

TERM,p otherwise

The label TERM,p must be accompanied by appropriate user input before the transformation is applied. The exact nature of the transformation applied is dependent upon the predefined rulename, but certain characteristics of the transformation may be generalized. The transformation results in either a single node replacement or a possibly many-leveled subtree; it may not generate siblings or a forest of siblings. As regards the deletion restrictions,

the subtree created by a predefined function is considered a single unit for editing purposes that is not subject to internal deletions or insertions. System provided predefined rules, if the input is valid, invariably result in a bound node or subtree of bound nodes; a free node in the subtree would imply knowledge of language-specific grammar rules which no general purpose predefined function could have. User-supplied predefined functions, allowable as a language-specific extension to the system, may admit such free nodes; however, the language implementor is responsible for ensuring the syntactic integrity of the AST is preserved over such transformations.

If the input accompanying the label is rejected by the predefined function, the transformation is null and the node is unchanged.

f. Undefined

Implicit Template:

NT,u => NT,u

The undefined label undergoes a null, implicit transformation.

4. User Attention

Of the eleven transformations, six define the action to be taken for the six possible nonterminal labels. The remaining five, the second transformation template for each of the five defined rule types, all require some form of

user attention prior to the application of the specified template. The form of user attention required is dependent upon the operator but generally may be characterized as consisting of two parts: an indication that the user wishes to direct attention to the current node, and a possibly empty character string utilized by the transformation as an input parameter. The five transformations requiring user attention fall into three classes, as follows:

a. IOPT, COPT, LOPT

The three optional operators require simply that the user elect to expand the optional node. Thus directing attention to an optional node is sufficient for application of the template and the character string parameter is not required.

b. ALT

The Alternation operator requires that the user, after directing attention to the alternation node, provide a character to be utilized in determining which of the possible alternatives is desired.

c. TERM

The TERM operator requires, in addition to the user's attention, a character string for processing by the predefined rule associated with the node.

The exact format of the user attention parameter is implementation dependent, but is summarized abstractly as follows, by operator:

operator	user attention
COPT	<elect option>
IOPT	<elect option>
LOPT	<elect option>
ALT	<char>
TERM	<string>

5. Deletion and Insertion

Earlier it was asserted that templates defined in accordance with an appropriate set of restrictions would allow deletion of any subtree from the AST using only the rulename of the subtree's parent node. We now verify that assertion based on the templates as defined above.

Of the six rule types, three may be excluded from consideration as potential parents of nodes to be deleted. Undefined rules never form children and thus are never referenced for deletion. Predefined rules are defined to create subtrees which can be edited only as complete units. Alternation rulenames never appear in bound nodes of the AST since the alternation rulename in a free node is overwritten with the rulename of the alternative rule chosen. Thus only concatenation, iteration, and list rules remain as potential parents of subtrees whose deletion is desired. The parent's rule type in each of these three cases may be positively identified by the parent node's operator: if the operator is ITER, the the parent rule is an iteration; if LIST, then it is a list rule; and if otherwise (either HEAD or a

predefined function), then the parent rule is a concatenation. The templates for these three rule types allow recreation of the original label which existed when the root node of the subtree to be deleted was initially created.

A parent concatenation rule, upon initial expansion, creates a fixed number of children, all of the forms NT,r and COPT,r. By inspection, no transformation or sequence of transformations on these labels for any of the six rule types may create additional siblings under the parent concatenation rule nor may they reorder the subtrees initially created. Thus the initial fixed number and order of children created remains constant. Suppose some subtree, say the ith, under the concatenation rule parent is selected for deletion. The sibling which was originally created by the concatenation rule as its ith child may be reconstructed by traversing the concatenation rule template until the ith sibling list element is encountered. This sibling list element contains the information by which the node replacing the subtree to be deleted may have its operator and rulename fields reinitialized. Deletion of a subtree under an iteration ru parent node is made possible by the consistent manner in which the two iteration rule templates create children of the parent node. The first child is created by the first template and the deletion process for the first subtree is similar to concatenation deletion. Subsequent subtrees, up to the trailing IOPT,i node, are created by the

second template and the information necessary to recreate any label may be retrieved from the first sibling list element of that template. The IOPT,i child is invariant in location and form and is not subject to deletion.

Deletion of the first subtree under a list rule parent is handled in the same manner as the first subtree under an iteration parent. Subsequent subtrees, up to the LOPT,1 node, are also similar to iteration rule subtrees except that they may have been created in pairs. Examination of the list rule's second template will reveal whether subtrees after the first must be treated in pairs or may be handled singly. In either event, the information necessary to recreate any given child is available in the template. The LOPT,1 child is not subject to deletion.

So far deletion has been concerned only with "unparsing" an incorrectly formed subtree to a single ancestor node so that the subtree may be correctly reconstructed. For subtrees of concatenation rules this is the only form of deletion which retains the valid derivation property. Subtrees of iteration rules, however, are all derived from the same label and thus are all syntactically equivalent when viewed from their root. Further, the only restriction on the number of iteration rule node subtrees is that there must be at least one in addition to the IOPT node. Thus, deletion of an iteration rule subtree, excepting throughout the trailing IOPT node, could be realized as the actual

physical deletion of the entire subtree including the root node, as long as at least one subtree remains. As a corollary, a node properly labelled in accordance with the iteration parent rule could be inserted in front of any node in the iteration sibling string without violating the valid derivation property. The insertion procedure requires the same information as deletion, the rule type and rulename of the parent node, in order to construct an appropriately labelled node for insertion into an existing iteration node sibling string.

List rules whose second argument is a terminal symbol form AST structures equivalent to iteration constructs and thus physical deletion (as opposed to unparsing to a single node) as well as insertion are valid operations. List rules in general present a more complicated problem in that subtrees after the first are formed in pairs. However, extending the argument concerning syntactic equivalence of subtrees to pairs of subtrees is straightforward and allows physical deletion and insertion to apply to list rule subtrees as well.

In summary, deletion is realized as a replacement operation for all concatenation rule subtrees and for solitary iteration and list rule subtrees, wherein the subtree to be deleted is replaced by a single node which is a reconstruction of the subtree's initial state. Under iteration and list parents where other subtrees exist, deletion

results in the physical removal of the subtree or subtree pair; reconstruction may be accomplished at the same or some other location under the parent by a separate insertion operation.

C. DISPLAY SCHEMAS

Thus far a method of constructing an AST has been developed utilizing transformations to expand nodes in accordance with a set of templates sorted by rulename such that the AST represents a valid derivation of the associated grammar. Attention is now focused on displaying the AST; in particular, a method is developed in this section by which the valid derivation of the grammar which the AST represents may be displayed.

Display of the AST is the result of a generalized inorder traversal, beginning with the root node, with terminal and nonterminal symbols being displayed in accordance with schemas associated with each label. The display need not be strictly preorder since provision is made to display subtrees under a parent node in any order as directed by the parent's rule schema. This capability is provided to allow for the case where the evaluator may have to access the subtrees in a different order than that implied by the syntax of the target language.

Schemas are referenced by the rulename associated with each bound and free node in a manner similar to the

referencing of templates so that the display associated with a subtree is independent of the context of that subtree.

The valid derivation need not be displayed in its entirety. For example, the means is provided to display all undefined nonterminals as they occur in the AST as part of the valid derivation. If the language implementor chooses, however, he may elect to not display any of the undefined nonterminals which appear in a partial grammar he is implementing in its incomplete state.

In the following two sections, first the schema language is defined and then the formation of schemas for each of the ruletypes is developed.

1. Schema Language

There are three types of display information provided for in the schema language: format control, literal strings, and subtree indicators. A system for handling comments has not yet been developed. However, it is envisioned as an extension to the schema language and not as part of the grammar for the target language.

Format control information is encoded mneumonically in the double capital-letter strings "NL", "TB", and "UT", interpreted respectively as "newline", "tab", and "untab". UT simply causes a variable, "tabcount", to be decremented. TB causes a tab control character to be transmitted to the output device and increments "tabcount". NL causes a newline character and "tabcount" tabs to be transmitted to the

output device. Format control information is provided for readability only.

Literal strings are arbitrary character strings, delimited by double quotes, that are transmitted directly to the output device. Literal strings provide the mechanism for the display of terminal and nonterminal symbols in the derivation represented by the AST.

A subtree indicator, denoted by a dollar sign followed by an integer interpreted as a child number, directs that that subtree be entirely displayed prior to resumption of display of the current schema. An optional display field, consisting of an equals sign followed by a literal string, may accompany the subtree indicator to provide the means for displaying undefined nonterminals, the three optionals, and TERM nodes, as described in the following paragraphs.

An undefined nonterminal may appear for a variety of reasons, the most common being as a placeholder in a partial grammar. Since the rule for the nonterminal does not exist, there can be no schema, so the optional field, if provided, is invariably utilized. If not provided, nothing will be displayed for the undefined nonterminal.

The three optional nodes, COPT, IOPT, and LOPT, require special handling since there is nothing inherently "optional" about a rule. Rather, the optional nodes are placeholders to indicate to the user the possibility that

the rule specified may be invoked, if the user so chooses, but also may be left uninvoked in a "complete" AST. Since it is the father rule which holds the information that this rule invocation may be an as yet unelected option, the father rule schem, contains the information, in the form of an optional display field, to display the node accordingly.

The predefined rule referenced by a TERM node is in general a language-independent system routine. As such, it has no knowledge of the nonterminal name which it, when invoked by the user on a string, is replacing in the valid derivation. Since the father rule does have this information, the father rule schema contains the optional display field necessary to properly display, within the context of the grammar, the rulename which the predefined rule will replace. In other words, this facility allows the language implementor to rename the predefined rule for display purposes.

When an option has been elected or a TERM node predefined rule has produced a bound node, both of which are displayable in their own right, the optional field associated with the subtree indicator is no longer necessary and will be ignored by the display algorithm. While these nodes remain free, however, the optional display field provides the user the information he needs to expand these nodes, as well as a logical symbol under which the GDE may place the cursor to indicate the current node.

A subtree indicator which may reference one of the three node types discussed above must, in order that a valid derivation be displayed, include an appropriate optional display field. The implementor may, of course, omit such a display field in which case nothing will be displayed for the node. In the case of an undefined nonterminal this may be the most pleasing result; in the case of optionals and TERM nodes such a display will not accurately reflect all free nodes in the AST that may be of interest to the user. The ommission of such an optional display field may be regarded under normal circumstances as a mistake in the language definition.

2. Rule-Specific Schemas

Construction of schemas is a straight-forward process when keyed to rule-type since the schema subtree indicators and literal strings must conform to both the R-ARGOT grammar rule definition and to the transformation templates associated with the rule definition in a consistent way. In the schema constructions which follow, format control information is ignored, but generally may be inserted into a schema any place that a terminal symbol is allowed.

a. Concatenation

Rule:

c : x1 x2 ... xn , xk = (rk | "["rk"]" | tk)

Schema:

cs : s1 s2 ... sn ,

	"tk"	if xk = tk
	$j="[rulename]"	if child j is optional
sk =	$j="<rulename>"	if child j is predefined
	$j="(rulename)"	if child j is undefined
	$j	otherwise

A single schema is required for the concatenation rule and may be constructed, if all nonterminals are realized as children in the order they are listed in the R-ARGOT rule, as follows:

Reading the R-ARGOT concatenation rule from left to right, for each symbol xk:

if xk is a terminal symbol, copy it to the schema as a literal string;

if xk is the jth nonterminal and is optional, write $j="[rulename]" to the schema;

if xk is the jth nonterminal and is predefined, write $j="<rulename>" to the schema;

if xk is the jth nonterminal and is undefined, write $j="(rulename)" to the schema.

if xk is the jth nonterminal symbol, and is not optional, undefined, or a predefined rule, write $j to the schema;

This algorithm for the construction of a concatenation schema is for the display of the entire valid derivation. If display of an undefined nonterminal, for example, is not desired, the subtree indicator for that child could either be written without the optional display field or be omitted entirely. While this algorithm assumes that the implementor wrote the concatenation template such that the children correspond in order to the nonterminals in the rule, this need not be the case. The schema must know the order, however, so that the display is an accurate representation of the derivation obtained from the grammar.

As an example of each of the possibilities listed above, consider the concatenation rule

simple : "program" name decls [externs] block "end" .

where the nonterminal "name" refers to a predefined function, "decls" is an undefined nonterminal, and "block" is a well defined, non-optional, non-predefined nonterminal. The schema for this rule, without any format control characters, would be

"program"$1="<name>"$2="(decls)"$3="[externs]"$4"end"

b. Alternation

Rule:

a : "(" char1:x1 "¦" char2:x2 "¦" ... "¦" charn:xn ")"

Schemas:

as1 : "{alternation rulename}"

as2 : "(char1:rulename1 ¦ ... ¦ charn:rulenamen)"

Since the transformations defined for an alternation rule are both single node replacements, the second one of which results in the alternation rulename being overwritten, it is clear that no semantic or schematic information required in a sentence in the language, as opposed to a valid derivation in general, may be associated with the schema for an alternation rule since once the alternative choice is made by the user, the rulename and thus access to the schema is no longer present in the AST. Thus the schema for an alternation rule could have been implemented as a subtree indicator optional field. We choose to provide a pair of explicit display schemas associated with the alternation rulename, however, to implement a "help" mechanism. The first display schema consists simply of a literal string composed of the alternation rulename in curly brackets and is the schema normally used to display the node. The second, optional at user request, is again simply a literal string but with the alternative rules and their associated keystrokes displayed in curly brackets.

For example, the following alternation rule

```
statement : { a:assignment ! c:conditional ! b:block }
```

would be displayed normally by the schema

"{ statement }"

or, if the user desired to see the alternatives and their keystrokes, by

"{ a:assignment ! c:conditional ! b:block }"

c. Iteration

Rule:

i : "+" r

Schemas:

is1 : $1

is2 : "[iteration rulename]"

The iteration (as well as the list) rules differ from concatenation in that they may have an indefinite number of children requiring display. Since no terminals are allowed in an R-ARGOT iteration rule and since every child is formed independently of the others in the sibling string, display of an iteration, while involving some work on the part of the display algorithm to traverse all of the subtrees one at a time, requires a pair of very simple schemas. The first is simply a subtree indicator used for display of all subtrees except the last. The subtree indicator may include an optional field for undefined and predefined rule display; from the transformation template definitions it is apparent that no child of an iteration node can be a concatenation optional node. The second schema is used for display of the last child, invariably an IOPT node.

d. List

Rule:

l : "#" r1 x "..." , x = { r2 | "[" r2 "]" | t }

Schemas:

```
ls1 : $1
          $1$2                    if x = r2
ls2 :     $1="[rulename2]"$2      if x = "["r2"]"
          "t"$1                   if x = t
ls3 : "[list rulename]"
```

The list rule requires three schemas in order to properly display the unique format the list structure conveys. Like the iteration rule, the list may have an indefinite number of subtrees; however, R-ARGOT allows the second argument to be a terminal symbol. Without this facility the inclusion of the list rule type is hardly justified since the most usual use of the construct is to separate grammatical entities with some punctuation mark.

The first schema is used for display of the first child. Subsequent children or pairs of children, depending on the specific list rule, up to the last in the sibling string, are displayed by the second schema. The display algorithm must keep track of which children it has displayed in traversing the list in order that this label schema structure display the sequence of subtrees correctly. The third schema is used for display of the last child, invariably an LOPT node.

As an example of the list rule schemas, consider the R-ARGOT rule

statements : # statement ";" ... .

The schemas generated to display this rule would be

ls1 : $1

ls2 : ";"$1

ls3 : "[statements]"

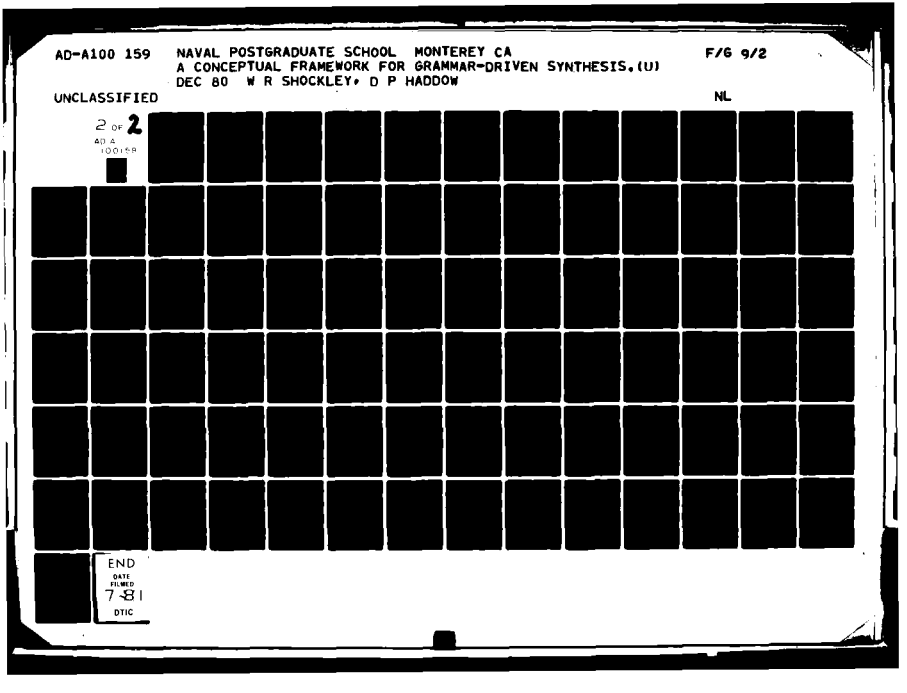
Note that a NL format control character would be appropriate after the ";" terminal in ls2 and before the literal string in ls3 in order to place each statement and semicolon pair on a separate line.

e. Predefined

A predefined display function should accompany each predefined rule scanner. The display algorithm will pass the subtree created by the predefined scanner to the named display function. For example, the predefined scanner "id" will scan an identifier, place it in the symbol table, and fill in the TERM node with the information allowing reference to that symbol table entry for the evaluator. On display, the routine "idout" will be called to cause the referenced identifier to be displayed.

D. THE LANGUAGE DEFINITION MODULE

The Language Definition Module is the grammatical database utilized by the Grammar Directed Editor in the construction and evaluation of an AST. The Language Definition Module has a fixed and an interchangeable component. The fixed component consists of the system predefined rules and


AD-A100 159
NAVAL POSTGRADUATE SCHOOL MONTEREY CA
A CONCEPTUAL FRAMEWORK FOR GRAMMAR-DRIVEN SYNTHESIS.(U)
DEC 80 W R SHOCKLEY, D P HADDOW
F/G 9/2
UNCLASSIFIED
NL
2 OF 2
AD A 100168
END
DATE FILMED
7-81
DTIC

functions. The interchangeable component, known as the Language Definition, is comprised of the language-specific grammar rules, templates, and schemas. In addition, the Language Definition may optionally include user-supplied predefined rules and functions supplementing or superceding those permanently installed in the system.

1. The Language Definition

The primary component of the Language Definition is the internal representation of the language-specific grammar as an ordered collection of grammar rules and their associated templates and schemas. The Language Definition, apart from user-supplied predefined rules and functions, consists of a Rule Tree and a string table. The string table contains the character string representation of the templates and schemas for each rule. The Rule Tree is the ordering mechanism for the grammar rules which provides access to the templates and schemas in the string table. The Rule Tree is a four-tiered hierarchy, the uppermost level of which is a head node for the tree. The next level consists of a sequence of head nodes, one for each defined grammar rule. Under each grammar rule node is a pair of head nodes, the first for the templates associated with the rule and the second for the schemas. The fourth, bottom-most tier consists of leaf nodes containing pointers to the template and schema strings stored in the string table. The regularity designed into the template and schema definitions for each

of the rule types allows accessing any leaf of the Rule Tree by the Editor utilizing only the operator and rulename information in an AST node label.

Appendix D is an Intermediate-Level Language Definition Grammar. Encoded by hand into a Language Definition as shown in Appendix E, the ILD Grammar provides the means to generate a Grammar Directed Editor for the construction of ASTs representing language-specific Language Definitions. When such an AST is evaluated by the predefined function ILD, the result is a language-specific Language Definition which may be installed in the Language Definition Module and utilized to construct applications-oriented ASTs in the language defined by the grammar. Appendix F presents a simple example of such an applications-oriented Language Definition from which ASTs representing strictly formatted memoranda may be constructed utilizing the GDE.

The ILD Grammar allows definition of grammars on an assembly-language level, i.e., many details which are computable from the R-ARGOT grammar rule must be entered by the user. For example, in the construction of an iteration rule the user is required to enter "rulename1" and "i-rulename" in a consistent manner throughout the formation of the templates and schemas. However, at this low level the mechanisms for checking such consistency do not exist. Thus the ILD Grammar is seen as a flexible but error-prone tool suitable for use primarily as a bootstrap mechanism for the

definition and implementation of a High-Level Language Definition Grammar which automatically derives as much information from the R-ARGOT rule as is possible. For grammars in which all nonterminal children of concatenation rules are to be created and displayed in the order listed in the rule, an extended R-ARGOT notation which provided the facility for inclusion of format control information and a means for specification of predefined functions as head nodes of concatenations would allow such automatic derivation. Development of such an extended notation as well as the corresponding HLD Grammar and function are deferred until the symbol table and evaluator designs are complete.

2. Predefined Rules

The set of system predefined rules provides the user a mechanism for entering strings representing simple, common constructs, such as identifiers and numbers, as well as more involved constructs, such as expressions, which even though composed of many parts and perhaps generating multinode subtrees in the AST, may be most conveniently viewed by the user as representing single logical units. Predefined rules are built-in, optional extensions to the Language Definition which provide the language implementor with a set of primitives upon which he may base his grammatical constructs. The set of predefined rules is modifiable and extensible by the language implementor through inclusion as an adjunct to the grammar definition a set of predefined rules which

supercede or complement the set permanently installed in the Language Definition Module.

Predefined rules may be viewed as a deviation from the grammar directed editing philosophy espoused throughout this work. The use of predefined rules allows the entry, after all, of syntactically incorrect strings which are not immediately, in the sense of character-at-a-time immediacy, detected and rejected as invalid. For example, compare a "pure", character-at-a-time grammar directed editor with a predefined rule augmented GDE on the terminal <string>, defined for illustration to be the concatenation of any characters except a space, and terminated by a carriage return. In the pure system, each character is examined and its validity checked as it is typed. In this example, if the user enters a string of valid characters and then a space, he is immediately informed that the space is unacceptable and is able to proceed without retyping that portion of the string thus far entered. The predefined rule system, however, would require that the entire string of symbols, including the incorrect space, be entered before rejecting it, and the user would have to retype the corrected string in its entirety.

We grant that grammar directed editing down to the smallest indivisible unit, the character, has a certain appeal. However, our predefined rule compromise is motivated by several advantages and mitigating arguments:

a. The time lapse between entering even a large predefined rule input string, such as a complex expression, and re-entering it if it is rejected as incorrect, is short.

b. The time lost in a predefined rule system in retyping the usually short input strings accepted by most predefined rules is offset by the time that would be lost in a pure system that requires control characters to guide the tree building via the language definition through the various alternatives involved in the larger grammatical constructs, such as expressions, that can easily be handled by predefined rules.

c. The syntactic integrity of the AST is always preserved by the system predefined rules since no change to the AST is made until the syntactic validity of the entire input string is confirmed.

d. Predefined rules simplify the language implementor's task by raising the level of the lowest grammatical constructs that must be defined in the grammar. Instead of having to work clear down to the character level, predefined rules provide as primitives the facilities for handling groups of characters, such as numbers, identifiers, and strings, which are the basic building blocks of data structures in general and programs in particular.

e. Given automatic lexical analyzer and parser generators, predefined rules for the class of grammatical constructs envisioned are easily built.

f. The suitable choice of predefined rules frees the language implementor from long-winded, needlessly detailed grammatical constructions for a wide variety of regularly-expressible productions. Grammars for language definitions, given such a set of easily understandable primitive constructions, would be more transparent and easier for the user to assimilate.

It is recognized that taking the predefined rule approach to its extreme limits could result in a compiler-like editor wherein huge segments are submitted for analysis to exceedingly complex predefined rules, thereby negating the benefits to be gained from a more rational grammar directed editing environment. However, within the guidelines presented here, the predefined rule approach has distinct advantages and leaves open avenues for exploration to the language implementor.

3. Predefined Functions

Nodes in the AST undergoing evaluation faall into one of three categories: undefined, head, and function. The class of undefined nodes includes all free nodes which may still exist in the AST. Head nodes nodes are the HEAD, ITER and LIST operator nodes created for synthesis of the AST, all of which are synonymous to the evaluator. Head nodes have no computational capabilities during the evaluation process but rather provide structure to the AST. Function nodes have as their operator one of the predefined

functions. Function nodes are generated by concatenation and predefined rules during synthesis of the AST and result in calls to the corresponding predefined function during evaluation. Function nodes may be leaves, as in nodes which reference symbol table entries, or they may be interior nodes. If interior, function nodes must have the number, order, and type of subtrees expected by the predefined function.

The set of predefined functions defines the range of computational power available to the evaluator and thus limits the capabilities available to the user of the GDE. A proposed set of system predefined functions, based on the primitives discussed throughout [Pratt,1975], is presented in Appendix G. This set of system functions may be augmented by the language implementor through additional or superceding function definitions included as extensions in the Language Definition.

IV. PROGRAMS AS DATABASES

A. INTRODUCTION

The material contained in this chapter was originally developed during the search for a solution to a particular problem: namely, that of storing the tree representation of the synthesized program in secondary storage, with complicated links to other data structures recorded in the leaves, in such a way that pointer and reference integrity could be maintained. This problem is aggravated by the consideration that such a stored structure might well be reloaded at a time when the physical contents of shared memory spaces currently in use by the system are quite different from the environment existing at the time that the tree structure was originally created.

Once this problem was recognized as being a database management problem, to which known techniques of database design were applicable, the solution was straightforward. The database design techniques described throughout this chapter are taken from [Kroenke 1977]. The relatively unorthodox view of programs as complex databases afforded by this insight, however, is of more general interest since it provides a new perspective on the nature of programming systems. In particular, these considerations provide some justification for the hope that grammar-driven tree

synthesizers are capable of building up a language-independent semantic structure.

B. PROGRAMS AS COMPLEX RELATIONSHIPS

In viewing programs as databases, we first recognize that the semantic contents of a program must be accessed by two entities: the human reader or writer, and the processor intended to execute the program. Comments excluded, the information available to these two entities is almost identical: that is, the human user can predict exactly the operation of the processor for a given program, and the processor deterministically executes the encoded intentions of the programmer. So without loss of generality, we may initially consider the program as a database accessed by the processor. In the case of a machine language program, the processor is the real machine on which the program is to execute. For a higher-level language, the processor is the hardware-software combination, or virtual machine, which is capable of translating and executing the program.

The "semantic content" of the program is the collection of potential evaluations which the processor may be required to perform throughout the course of execution. For the moment, we disregard the order of execution. Each evaluation consists of the selection of one of many primitive operations which the processor is capable of performing, and the application of that chosen primitive

operation to a number of arguments, contained in one or more registers, or memory locations addressable in some way.

Upon reflection, it is clear that both the set of primitive operations and the set of addressable memory locations are databases in their own right. The keyname, or code by which an entry can be uniquely located, for the set of primitive operations is the operation name, or opcode, and that for the collection of potential arguments is the address.

Clearly, the set of potential evaluations is, in the terminology of database theory, a complex relationship between primitive operations and registers. A given operation may be applied to many different sets of arguments within the course of a program execution, and a given register may be the argument for a number of different operations. There is no functional relationship between items of the two databases in either direction, which means that neither keyname can be used to uniquely identify an item in the complex relationship between them.

C. DECOMPOSITION OF THE EVALUATION RELATION

Standard database design techniques specify several ways by which each of the elements of a complex relationship between two databases can be referred to in a systematic and unambiguous way during database access. Two general methods of approach are used. One is to (arbitrarily) force the

relationship to be simple (many-to-one in one direction only), by rejecting from the allowed range of possibilities any members of the relationship which would cause the relationship to be complex. In this case, the keyname for one of the underlying databases can be used to unambiguously refer to members of the relationship as well. The second method is to decompose the relationship into two simple relationships by constructing an intersection database.

There exist programming systems in which the first strategy is adopted. For instance, if the restriction is made that registers may not be re-used, so that at most one, and only one, primitive operation is applied to a given register, a purely functional, or no-assignment programming system is obtained. In such a system, the only named semantic elements are functions and constants (which may be regarded as functions). Registers need not be named since whenever one is needed, it can be drawn from a pool, used once, and discarded by the processor.

This approach is considered mathematically elegant, but it is not much in use in non-academic programming systems.

In the second approach, an intersection database is created, consisting of one entry for each distinct member of the complex relationship. As a minimum, in order to allow reference to the generating databases, each entry in the intersection database must contain the keynames for those entries in the original data sets with which it is

associated. Thus, for a programming notation, each entry in the intersection database must contain, at a minimum, an opcode and a register address for each argument, in some form.

The archetypical entry for the intersection database corresponding to the evaluation relationship is thus:
OPCODE ADDRESS(1) ADDRESS(2) . . . ADDRESS(N) .
This format is recognizable as the atomic unit of notation for most common programming systems, from machine code to high level languages. Each single such entry corresponds to what is normally referred to as an instruction. In summary, we assert that a program is nothing more than the intersection database for instances of the evaluation of accessable operands by the primitive operations available to the evaluating processor.

D. CONTROL STRUCTURE

We have heretofore ignored the question of how the order of execution of the evaluations is to be specified within the program (the basic elements of which are now seen to be entries in an intersection database). This order corresponds to the logical access sequence of the set of instructions. Thus, we may equate the ordinary notion of the control structure of a program, to the database-oriented notion of a logical access structure for the program database. The simplest access mechanism for a database is

to order it as a simple sequence. Under this protocol, the elements of the database will be presented to the accessing entity in a strictly invariant sequence.

Such an accessing structure is realized in such simple programming systems as that of a keystroke-programmable calculator. A sequence of keystrokes can be entered and automatically reproduced at will, but there is no possibility of automated branching.

Such programming systems are fundamentally limited in mathematical computational power. The simplest modification to such an access regime is to allow conditional branching, so that a part of the instruction sequence may be repeated or skipped, based on the contents of a register at the time the branch is reached.

Machine and assembly-level programming systems, as well as such high-level languages as BASIC and FORTRAN, are organized on such a plan.

E. STRUCTURED PROGRAMMING SYSTEMS

The disadvantage of a sequential access mechanism is that the resulting database does not have local integrity. Instruction sequences which may be logically adjacent under certain circumstances are not necessarily physically adjacent. This access organization presents no real disadvantages for the machine processor with a random-access architecture, but can be quite confusing for the human

programmer. To render the program database more accessible to the user, the notion of structured programming was developed. This organizational technique consists of organizing the access of a program database in a hierarchical (tree-like) manner, so that program control follows a hierarchical program structure which can be expressed as a string generated by a context-free grammar (and thus has an associated physically hierarchical structure induced by the grammar). Such program control facilities as functions and subroutines were the earliest "structured constructs". The syntax of such languages as PASCAL and ALGOL, however, were consciously designed to facilitate the expression of a hierarchical control structure, and make the expression of a disordered, sequential control structure less attractive than the use of "structured" control operators. It is this historical development which encourages us to hope that a language-independent semantic tree structure may be built using a grammar-driven tree editor. Basically, we note that it has become a conscious design principle in the development of structured programming languages, to ensure that program control flow follows the syntactic organization of the language. The underlying set of primitive operators have a great deal in common. Language-dependent primitives can be added to the set available to the processor and evaluated without regard to the specific syntax by which they are

expressed, provided that the overall control structure of such additional primitives is also hierarchically organized.

F. PHYSICAL REPRESENTATION OF A TREE-STRUCTURED PROGRAM

We are left with the problem of physically representing a tree-structured program in a sequentially organized physical memory space. The problems encountered are precisely those encountered when attempting to implement any hierarchically organized intersection set. They stem from the requirement to refer, directly or indirectly, to the entries in the parent databases from more than one place in the intersection database. Two general strategies, each with its own advantages and disadvanteges, are currently in use in database management systems.

1. *Sequential Tree Representation*

This strategy is implemented by representing the tree as a linear list of nodes and their contents in preorder sequence. References to the parent databases are embedded in the listing by keyname. The complexity of the relationship implies that each such keyname must be repeated many times throughout the list. Special delimiters are used between node listings to indicate whether the next node is a child, sibling, or uncle of the last. If one of the keynames is to be changed, a search of the listing must be made to find all of its occurrences. A second major disadvantage is that in order to access any part of the

list, the list must be traversed sequentially from the beginning. On the other hand, no pointers need occur anywhere in the list, so that it can be moved about freely from one place to another without change.

2. Linked Representation of Trees

Trees are represented in this strategy by nodes linked together using pointer fields within each node. A pointer is either the absolute address of the entity pointed to, or an offset or array subscript which can be used by routines in the system to calculate such an address. The salient feature of a pointer reference is that it allows reference by some mechanism which is independent of the value of the referenced entity. Thus, the value of the entity itself can be changed without changing all of the references to it, which are still valid (provided, of course, that the change is made without physically moving the changed record.) When the tree itself is represented by means of nodes linked with pointers, it is common to link the leaves of the tree to the parent databases with pointers as well. It is assumed that a means exists to distinguish such external links from the internal links defining the tree structure itself. This representation has as one major advantage the ability to be quickly traversed (by following pointers). Another major advantage of this strategy is that information in referenced databases need only be recorded once, and can be changed without updating any pointers.

Deletion of information is somewhat more difficult, but can be accomplished by constructing and maintaining cross-reference lists (inverted lists) which contain pointer references to all nodes in the tree referring to a given record in the parent database. The primary disadvantage of such a representation is that the structure cannot be moved or stored without a great deal of pointer modification. The use of relative pointers is an inadequate solution, since the consistency of references to the parent databases, which need to be moved and managed as separate entities, must still be maintained.

3. A Hybrid Strategy for Tree Representation

An examination of these characteristics indicates that the linked representation is preferable when changes are to be made to either the parent or tree databases, but that the sequential representation is preferable when the database is to be transmitted from one location to another, or stored unchanged for a relatively long period of time. (Storage is equivalent to transmission from one time to another, and is thus logically the same problem as that of movement.)

We conclude that the linked representation is an appropriate representation for the program tree during synthesis and evaluation, but that the program tree should be moved (or stored on secondary storage) in sequential, pointer-free format. Links to the parent databases are

converted from pointer references to reference by keyname. The next section addresses the problem of how conversion between the two representations can, in general terms, be accomplished.

G. PROCEDURAL REPRESENTATION OF DATA

In order to incorporate these ideas into a feasible design, we consider the facilities that would have to exist in such a system. Since the program tree is to be operated on in main memory with a linked representation, we may assume that a data manipulation package exists which is capable of synthesizing and maintaining all of the pointers required to keep the linked structures coherent and consistent. Consider the process of removing a sequentially organized tree structure from secondary storage and loading it into internal memory. This process must consist of ordering a particular series of function activations with particular arguments from the data manipulation package, causing the desired structure to be built within physical memory. The sequential representation is seen to be nothing but a program for the data manipulation package, which is itself a processor with a number of primitive operations.

Moreover, a strictly sequential control protocol for this program is possible, given a reasonably powerful set of primitives in the data manipulation package, since a tree can be synthesized in strict pre-order sequence (the parent

for each child exists at the time of the child's synthesis.)

We conclude that the appropriate secondary representation for a program tree is as a sequential list of instructions, to be translated by some simple interpreter into a series of calls to the data manipulation package.

The offload, or transmit process, consists of a pre-order traversal of the linked representation, emitting the appropriate instructions for recreating the skeleton of the tree and filling in the contents of each node as it is reached. At the same time, references can be removed from the appropriate cross-reference lists, triggering removal of the data item from the parent database when a reference count of zero is reached. During onload, the skeletal structure of the tree is recreated, and external references in symbolic form reloaded into the appropriate parent database. Pointer and cross-reference list creation and maintenance is performed automatically by the pre-existing data manipulation package.

The secondary representation can thus be viewed either as data, representing the tree in linear format, or as a program for the data structure manipulation package which will cause a logically equivalent tree to be reconstructed in available memory.

As a beneficial side effect, if the capability is installed to allow the onload and offload translators to read to or from strings in main memory, the described system

provides an easy way to copy or move subtrees, as well as to encode tree-building templates efficiently. In fact, the proposed mechanism becomes the method of choice for any and all movement of tree structures from one location or time to another, since the data in the transmitted stream is entirely logical, containing no reference to any implementation details. The process would even allow internal representations to be transmitted from one installation to another with a completely different implementation, since all implementation-dependent data is removed during the offload process and reinserted during the onload process.

H. SUMMARY

In this section we have viewed programs as specialized databases, and have found that standard database models correspond nicely to various programming language styles. Two fundamental conclusions have been reached. The first is that it seems very likely that grammar-driven tree editors can be used to produce trees representing the control structures for common programming languages in a syntax-independent, directly-evaluable format. This hope is based on the direct expression of hierarchical control structures by the syntactic hierarchy implicit in the defining grammars of current programming languages, and the recognition that a small set of such control structures provides the common

base for current language design.

The second result is the solution to a technical problem: that the appropriate format for such program trees is in linked form when the tree is undergoing modification, and as a sequential, procedural, pointer-free list of instructions when the tree is being stored, or transmitted from one point to another.

V. A PROTOTYPE SYSTEM DESIGN

In this section, the design for a prototype system demonstrating the feasibility of the ideas developed in previous chapters is described. Since the implementation of the described system is, at present, incomplete, the design is presented only in broad outline. A full description of the demonstration prototype will be provided as a Technical Report when the initial implementation is complete.

The approach taken is to first describe a complete system for a grammar-driven, language independent programming environment, and then select a subsystem for implementation as a prototype feasibility study. The prototype subsystem will be used to generate statistics concerning memory size and computational efficiency, as well as to refine the user interface, with the possibility remaining of extending the prototype to a more complete implementation at a future time.

A basic block diagram of the complete system is provided as Figure 5.

A. SYSTEM MODULES.

The proposed system consists of the following modules:

1. Data Structure Support Module.

This module contains packages of functions, each package implementing a specific abstract data type needed by the remainder of the system. At a minimum, the abstract data type packages needed include one supporting an indefinite number of indefinitely large association lists, (to represent the contents of tree nodes), and one supporting general ordered trees, optimized toward reasonably efficient traversal in all directions. In addition, the tree support package must include a facility for linking the leaves of trees to other data items, such as strings, symbol table entries, numerical contents, and so on. Each tree node (internal as well as leaf) must be linkable to an association list representing the contents of the node.

In addition to supporting tree and association list data types, this module is responsible for supporting any additional data types for which the need arises and which are not supported directly by the language used for implementation. (In particular, the implementation currently being developed requires a very primitive string table which serves as a rudimentary symbol table.)

2. Grammar-Driven Environment Module.

This module provides an editor-like interface for the user. It translates user commands into appropriate system actions, which include editing functions, directives

to evaluate a particular program structure, and movement of Abstract Syntax Trees from secondary to primary storage and back again. A major component of this module is the grammar-driven synthesizer itself.

3. Memory Management Module.

This module comprises the actual system primary memory itself, which is used to store the LD (Language Description) and AST (Abstract Syntax Tree) currently in use. In addition, the primary memory module contains the data structures being manipulated by the Data Structure Support Module.

4. File Management Module.

This module implements a single-user workspace on secondary storage which contains all of the LD's available to the user, as well as all of the AST's which may have been previously created and saved. These components are stored in sequential, pointer-free format as discussed in Chapter IV.

5. Input/Output Manifolds.

These modules manage the system input and output streams, which may be redirected as required by components of the system (including the user) to various physical devices. The input stream may be taken from the keyboard, a file on secondary storage, or a string in primary storage. This assignment may be changed dynamically during the operation of the system. Similarly, the output stream may

be dynamically directed to the CRT, a string in primary storage, or to a file on secondary storage. (The term "manifold" is used to suggest that these functions may be thought of as three-position switches, the setting of which may be changed at will during system operation.)

6. Onload and Offload Translators.

These modules, controlling the Data Structure Support facilities, convert the sequential data representations stored on secondary storage to the linked representation needed when an LD or AST is loaded into primary memory, and vice versa. As a secondary feature, since the input and output streams may originate or be directed to internal strings, these modules can be used to "quote" or "unquote" tree structures, as when a template is translated into an actual subtree replacement.

B. PRE-EXISTING MODULES.

The current implementation is being made using the C Programming Language on a PDP-11 with the UNIX Operating System. (UNIX is a trademark held by Bell Laboratories, Inc.) This software combination provides a C-accessible interface to memory and file management facilities. In addition, a complete library of string handling and input/output functions is available. In consequence, the memory and file management modules described above may be thought of as already in existence, for the purpose of

describing the prototype subsystem. In addition, keyboard and CRT interfaces are already operational: under the UNIX operating system, hardware interfaces are mapped into the system as files with conversion routines provided transparently. Thus, for the Input/Output Manifold module we need only provide a means of diverting the input and output streams from one file to another, or to main memory.

C. SUBSYSTEM SELECTION.

Given the broad outline of system module function provided above, a minimally capable prototype subsystem can be selected for initial implementation. Such a subsystem must be capable of initialization, synthesis, display and storage of an AST in order to demonstrate convincingly the feasibility of the concepts outlined in previous chapters. Facilities to evaluate (execute), revise, and debug previously entered AST's may be deferred, as may the facility to easily install a new Language Definition. Therefore, the capabilities provided by each of the modules in the prototype subsystem may be redefined as follows:

1. Data Structure Support Module.

Full packages supporting general ordered trees and association lists are needed. In addition, a primitive capability to store and reference string values is needed. The capability to support sophisticated symbol table structures may be deferred to such time as semantic

information is needed to allow execution of AST structures.

2. Grammar-Driven Environment Module.

The only major capability required by the prototype subsystem is the "append" function, which can be used to create AST structures. In addition, a working display mechanism with simple cursor control facilities is needed. A frame-oriented display mode is satisfactory for the prototype system (although eventually a screen-oriented display driver would be desirable). Finally, facilities for storing and retrieving AST's to and from secondary storage as well as a facility (however cumbersome) for installing new language definitions is needed.

3. Input/Output Manifolds.

These modules need to be implemented in full, in order that secondary storage may be used, and in order to allow templates existing in primary memory to appear in the input stream for processing by the Onload translator.

4. Onload and Offload Translators.

These components also must be fully implemented for the same reason as the Input/Output Manifolds. The implementation must be flexible enough so that as more sophisticated data structure packages are added, the sequential representation can syntax can be extended to accomodate onload and offload of keyfields in the new structures.

5. Bootstrap Procedure.

The system can be initialized as follows. We currently regard Language Definitions as being written in one of three languages, or notational systems: a high-level format (which is to consist of R-ARGOT notation with display and semantic specification extensions), intermediate-level, (the notation developed in Chapter III), and low-level, (the sequentialized, pointer-free representation of an internal tree corresponding to the desired LD, using the language alluded to in Chapter IV.).

There is no fundamental difference between the intermediate and low-level formats, since they represent two alternative representations for the same database. Translation from one format to the other is performed automatically by the onload and offload translators when this database is moved to and from secondary storage.

In order to bootstrap the system, once all of the modules have been compiled and linked, it is necessary only to perform the job of manually translating an intermediate-level description of the intermediate-level language to the corresponding low-level description, and install the resulting text as a file accesible to the system using a conventional editor.

At this point, the system facilities can be actuated to load the file as a language descriotion into system primary memory. During the load, the onload translator will

convert the description into a linked representation of the database needed to describe and guide the synthesis of new language descriptions in the intermediate format. That is, the system itself can now be used to create, as a grammar-driven editor, additional language descriptions.

VI. SUMMARY.

A. CONCLUSIONS.

In the preceding chapters, a conceptual foundation for the interactive creation of databases, structured hierarchically according to a given context-free grammar, has been provided. The primary conclusions supported by this work are:

1. A basic model for the described process is that of a valid sentential form generator, rendered determinate by allowing for the interactive selection of which production to apply and at which point in the already-derived structure the selected substitution is to be made.

2. Notations exist (e.g. the R-ARGOT notation) for the specification of general, context-free grammars which are both human-oriented and directly interpretable as the knowledge base for such a system.

3. The basic mechanism correctly interprets ambiguous or incomplete grammars, as well as allowing for the synthesis of correctly labeled incomplete derivations.

4. Analogous mechanisms can be described which derive and display not strings, but derivation trees which are morphisms of validly derived strings under the specified grammar.

5. The grammatical notation can be transformed into context-independent operation codes with arguments which can be stored in the leaf nodes of the derived tree in such a way that subsequent synthesis proceeds correctly, and subtree deletion can be efficiently and consistently performed without examination of the surrounding context in the tree.

6. The resulting derivation trees can be used to encode semantic information in such a way that the trees can be evaluated correctly without further reference to the syntactic, as opposed to physical, structure of the tree. (This assertion is a speculation, not a firm conclusion.)

7. A method exists for storing such structures in such a way that their consistency does not depend on any external data structures save the language definition itself.

B. WORK IN PROGRESS

Implementation of the prototype subsystem is currently in progress, with no difficulties currently foreseen. The only module awaiting final coding and test is the Grammar-Driven Environment module itself, and the algorithmic specification of the functions needed has already been accomplished. Provided that no further difficulties are encountered, a complete description of the prototype subsystem will be later provided as a Naval Postgraduate School Technical Report.

The prototype subsystem code is oriented toward a demonstration of technical feasibility as opposed to storage or execution time efficiency. However, it has been written in a highly-modularized manner, so that after instrumentation and performance measurements appropriate modifications can be made fairly easily. An attempt has been made to provide for the extension of the prototype system to a more complete realization of the original system design.

C. FUTURE RESEARCH DIRECTIONS.

After completion of the prototype subsystem, two directions are indicated for future investigation.

1. Extension of the Prototype Subsystem.

a. Symbol Table Implementation.

A generalized symbol table data type must be defined which will adequately support a wide range of programming languages.

b. Semantic Action Implementation.

A class of primitive operations (including access facilities to the defined symbol table structure) must be formulated, provision made for language-implementer definition of additional primitives, and an AST interpreter written.

c. Pattern-Matching.

A pattern-matching facility should be provided as part of the user interface as a sophisticated means of cursor control. A fairly simple pattern-matching capability, when combined with the pre-existing capability to access the AST in a syntax-oriented way, would allow the user to search and access the structure in very sophisticated ways; e.g. such commands as "find the next occurrence of an assignment to identifier a" could easily be formulated. Moreover, when combined with a relatively straightforward debug facility, (for example, setting of break-points) a very high-level program test facility could be provided.

d. High Level Language Descriptions.

The high-level format for both syntactic and semantic language specification should be formulated and implemented as a more convenient means for implementing new languages.

e. Debugging Tools.

Provisions should be made to allow the user to set breakpoints, access the current data environment, and order step-by-step execution modes from the editor.

f. Dynamic Language Changes.

The feasibility of allowing language changes to be made dynamically during AST creation or execution at points specifiable in the language definition should be

investigated. Related to this problem is the provision of a facility to link (perhaps dynamically) one AST to another.

g. Increased Storage Efficiency.

Once basic design parameters, now indefinite, (Such as number of primitive operations) are made final, the desirability of packing data fields into AST nodes rather than using the space-inefficient association list implementation, and the resulting impact on time-efficiency, should be studied.

h. Full User Interface.

Deferred edit functions, such as delete and insert, should be installed in the Grammar-Driven Environment Module.

2. Additional Applications for the Technology.

The conceptual framework provided by this paper is sufficiently general to support unexpected applications in areas quite distant from the field of programming environment design. A few such applications are suggested below:

a. Generalized Editing.

Generalized editors, as described in [Fraser 1980], are editors which provide for the manipulation and display of data structures other than text files. The mechanism is well-suited for the direct editing of a hierarchically organized database of any type.

b. Sparse Programming Languages.

Current programming languages are designed with a parser-based implementations as a fundamental assumption. For that reason, they typically include many keyword and punctuation symbols which are irritating, because superfluous, to human users. Because the described technology can utilize ambiguous grammars, sparse languages with the minimum amount of punctuation needed for human comprehensibility can be described which could be implemented using grammar-driven synthesis as the fundamental input mechanism. In fact, improved performance from the synthesizer could be expected for such a "pseudo-code"-like language, since the inherent semantic density of the derivation tree could be made very high.

c. Artificial Intelligence Applications.

In the described design, considerable pains have been taken to provide a simple, uniform method for grammar rule and point of application selection, suitable for use by a human operator. There is no fundamental reason why very complicated heuristic methods could not be used, however, to select the rule to be applied and the place in the current structure the application is to be made. For instance, a production system (in the Artificial Intelligence sense) could be used to perform this function. The resulting hybrid system would have a heuristic front end, and an algorithmic back end, with the desirable property that

whatever structure the heuristic front end attempted to build, the resulting structure would always be guaranteed to be correct in terms of the "deep structure" specified by the language description. Attempts by the heuristic module to perform inconsistent modifications would be detected, prevented, and reported by the synthesis module. A knowledge representation based on such a system would be able to interact with the user in very irregular, and occasionally incorrect, ways, while preserve a fundamental internal database with guaranteed consistency.

APPENDIX A. NOTATIONAL SYSTEMS FOR CONTEXT-FREE GRAMMARS

1. BACKUS-NAUR FORMAT (in R-ARGOT)

```
context-free-grammar: + production .

production: non-terminal "::=" [ right-hand-side ] "." .

right-hand-side: + construct .

construct: ( terminal ¦ non-terminal ) .

non-terminal: "<" string ">" .

terminal: "string".
```

We assume that "string" is a sequence of any appropriate character set not including the metasymbols.

Note that this notation is in itself a regular language.

2. ARGOT NOTATION (in R-ARGOT)

```
ARGOT: + rule .

rule: rule-name ":" concatenation.

concatenation: +sub-expression .

sub-expression: { optional-iteration
                ¦ simple-iteration
                ¦ list-iteration
                ¦ option
                ¦ alternation
                ¦ optional-alteration
                ¦ rule-name
                ¦ terminal
                ¦ group
                } .

optional-iteration: "*" sub-expression .

simple-iteration: "+" sub-expression .

list-iteration: "#" sub-expression sub-expression ". . . " .

option: "[" concatenation "]" .
```

```
alternation: "(" concatenation "¦" alternatives ")" .

optional-alternative:"["concatenation "¦" alternatives"]" .

alternatives: # concatenation "¦" . . .   .

group: "(" concatenation ")" .

terminal: " " " string " " " .

rule-name: string.
```

("string" is taken to be a predefined rule.)

3. R-ARGOT (in R-ARGOT)

```
R-ARGOT: + rule .

rule: rule-name ":" expression "." .

expression: { concatenation
            ¦ iteration
            ¦ list-iteration
            ¦ alternation
            } .
concatenation: +field .

iteration: "+" rule-name .

list-iteration: "#" rule-name field ". . . " .

alternation: "{" rule-name "¦" alternatives "}" .

alternative: # rule-name "¦" . . .   .

field: { rule-name
       ¦ option
       ¦ terminal
       } .

option: "[" rule-name "]" .

terminal: " " " string " " " .

rule-name: string .
```

Note that this notation is, in itself, a regular language.

APPENDIX B. A GRAMMAR FOR PASCAL IN R-ARGOT

```
PASCAL: "program" identifier "(" name-list ")" ";"
          block "." .

block: [ labels ][ constants ][ types ][ variables ]
       [ subroutines ] "begin" statements "end"  .

labels: "label" integers ";" .

constants: "constant" c-decls ";" .

types: "type" t-decls ";" .

variables: "var" v-decls ";" .

subroutines: + s-decl .

integers: +integer .

c-decls: # c-decl ";" . . .  .

c-decl: identifier "=" constant .

t-decls: # t-decl ";" . . .  .

t-decl: identifier "=" type .

v-decls: # v-decl ";" . . .  .

v-decl: name-list ":" type .

name-list: # identifier "," . . .  .

s-decl: {  p-decl
        |  f-decl
        }.

p-decl: "procedure" identifier [ parameters ] ";"
            block ";" .

f-decl: "function" identifier [parameters] ":" identifier";"
            block ";" .

parameters: "(" param-list ")" .

param-list: # param-section ";" . . .  .
```

```
param-section: {  f-params
               ;  v-params
               ;  p-params
               ;  c-params
               }.

f-params: "function" name-list ":" identifier .

v-params: "var" name-list ":" identifier .

p-params: "procedure" name-list .

c-params: name-list ":" identifier .

type: {  scalar-type
      ;  subrange-type
      ;  pointer-type
      ;  set-type
      ;  array-type
      ;  record-type
      ;  file-type
      ;  identifier
      } .

scalar-type: "(" name-list ")" .

subrange-type: constant ".." constant .

pointer-type: "↑" identifier .

set-type: [ packed ] "set" "of" simple-type .

array-type: [packed] "array" "[" subscripts "]" "of" type .

record-type: [ packed ] "record" [ field-list ] "end" .

file-type: [ packed ] "file" "of" type .

packed: "packed" .

simple-type: {  identifier
             ;  scalar-type
             ;  subrange-type
             }.

field-list: {  var-fields
            ;  mixed-fields
            } .

mixed-fields: fixed-fields [ and-var-fields ] .
```

```
and-var-fields: ";" var-fields .

fixed-fields:  # fixed-field ";" . . .  .

fixed-field: name-list ":" type .

var-fields: "case" [ tag ] identifier "of"
                    variants .

variants: # variant . . .   .

variant: constant-list ":" "(" [ field-list ] ")" .

constant-list: # constant "," . . .  .

statements: # statement ";" . . .  .

statement: [ integer ][ action ] .

action: {  assignment
        ;  procedure-call
        ;  compound
        ;  if-statement
        ;  repeat
        ;  while
        ;  for
        ;  case-statement
        ;  goto
        ;  with
        } .

assignment: variable ":=" expression .

procedure-call: identifier [ arguments ] .

arguments: "(" arglist ")" .

arglist: # argument "," . . .   .

argument: {  identifier
          ;  expression
          } .

compound: "begin"
              statements
          "end" .

if-statement: "if" expression "then"
                    statement
              [ else-part ] .
```

```
else-part: "else"
                statement .

repeat: "repeat"
             statements
          "until" expression .

while: "while" expression "do"
            statement .

for: "for" identifier ":=" expression t-or-d expression "do"
          statement .

t-or-d: (  downto
              |     to
              ) .

downto: "downto" .

to: "to" .

case-statement: "case" expression "of"
                          cases
                  'end" .

cases: # case ";" . . .   .

case: constant-list ":" statement.

with: "with" variables "do"
            statement .

goto: "goto" integer .

variables: # variable "," . . .  .

expression: (   lt
            |  lte
            |   eq
            |  gte
            |   gt
            |  neq
            |  in
            |  s-expression
            ) .

lt: s-expression "<" s-expression.

lte: s-expression "<=" s-expression.

eq: s-expression "=" s-expression.
```

```
gte: s-expression '>=" s-expression.

gt: s-expression ">" s-expression.

neq: s-expression "<>" s-expression.

in: s-expression "in" s-expression.

s-expression: [ sign ] u-expression.

sign: {  plus-sign
      ;  minus-sign
      }.

plus-sign: "+" .

minus-sign: "-" .

u-expression: {  plus
              ;  minus
              ;  or
              ;  term
              } .

plus: term "+" term .

minus: term "-" term .

or: term "or" term .

term: {  times
      ;  quot
      ;  div
      ;  mod
      ;  and
      ;  factor
      } .

times: factor "*" factor .

quot: factor "/" factor .

div: factor "div" factor .

mod: factor "mod" factor .

and: factor "and" factor .
```

```
factor: {  group
        ;  not
        ;  set
        ;  v-or-c
        } .

group: "(" expression ")" .

not: "not" factor .

set: "[" [ set-members ] "]" .

set-members: # set-member . . .   .

set-member: {  range
            ;  expression
            } .

range: expression ".." expression .

v-or-c: {  unsigned-constant
        ;  variable
        } .

variable: identifier [ modifiers ] .

modifiers: + modifier .

modifier: {  subscript
          ;  field-reference
          ;  indirection
          } .

subscript: "[" expressions "]" .

field-reference: "." identifier .

indirection: "↑" .

expressions: # expression "," . . .   .
```

It is assumed that predefined input scanners exist for the rule names "integer", "identifier", "constant", and "unsigned-constant".

APPENDIX C: TRANSFORMATION TEMPLATE GRAMMAR

The following grammar defines symbol strings which are interpreted as calls to tree-building and node-modifying routines whose existence is assumed, as is the interpreter which makes those calls. Also implicit in the following definitions and discussion is the notion of a "current node", defined for the purpose of the application of templates to be any free node in an AST.

```
template:   { subtree | siblist } .
subtree:    boundnode [childlist] .
childlist:  "(" siblist ")" .
siblist:    # freenode ";" ... .
boundnode:  boundop rulefield .
freenode:   freeop rulefield .
rulefield:  "," rulename .
boundop:    { HEAD | ITER | LIST | pdf } .
odf:        { (predefined functions) } .
freeop:     { NT | ALT | COPT | IOPT | LOPT | TERM } .
rulename:   { (grammar rulenames)
            | (predefined rulenames) } .
```

The Template Grammar produces operator and rulename pairs, both bound and free, punctuated by the terminal symbols "(", ";", "," and ")" which are interpreted as follows:

"(": Create a child node under the current node, make the node created the current node, and overwrite the OP field with the operator listed next.

";": Create a right sibling of the current node, make the node created the current node, and overwrite the OP field with the operator listed next.

",": Overwrite the RULE field of the current node with the rulename listed next.

")": Make the father of the current node the new current node.

The first symbol of every template is an operator, either free or bound, which overwrites the OP field of the current node. The current node is the only node in the AST which is modified in any way by a template; new nodes may be created, but always within the context of the current node.

The templates defined by this grammar allow definition of the transformations in Chapter III. The following examples illustrate the various constructions most commonly encountered.

1. Single node replacement, rule field unchanged:

Transformation:

NT,a => ALT,a

Template:

ALT,a

2. Single node replacement, operator and rulename modified:

Transformation:

ALT,a => NT,r

Template:

NT,r

3. Replacement with sibling string:

Transformation:

IOPT,i => COPT,r2 NT,r1 IOPT,i

Template:

COPT,r2 ; NT,r1 ; IOPT,i

4. Replacement with subtree:

Transformation:

NT,c => NT,r1 COPT,r2 NT,r3

Template:

HEAD,c (NT,r1 ; COPT,r2 ; NT,r3)

APPENDIX D: INTERMEDIATE-LEVEL LANGUAGE DEFINITION GRAMMAR

```
ILD:        langname rulelist [extensions].
rulelist:   + rule.
rule:       { c-rule
            ¦ a-rule
            ¦ i-rule
            ¦ l-rule }.
c-rule:     { c-rule-a
            ¦ c-rule-b }.
c-rule-a:   c-rulename ":" cdef-a
                  "=>" ctla "=>" csla.
cdef-a:     + defpart.
defpart:    { rulename ¦ option ¦ terminal }.
option:     "[" rulename "]".
ctla:       headop "(" freelist ")".
headop:     { head ¦ pdf }.
head:       "HEAD".
pdf:        { (predefined functions) }.
freelist:   # freenode ";" ... .
freenode:   freeop "," rulename.
freeop:     { nt ¦ copt }.
nt:         "NT".
copt:       "COPT".
csla:       + dispart.
```

```
dispart:    { subtree ¦ literal ¦ format }.
subtree:    "$" integer [opdisfld].
opdisfld:   { optodf ¦ pdfodf ¦ undodf }.
optodf:     "=""[" rulename "]""".
pdfodf:     "=""<" rulename ">""".
undodf:     "=""(" rulename ")""".
c-rule-b:   c-rulename ":" cdef-b
                 "=>" ct1b "=>" cs1b.
cdef-b:     terminal.
ct1b:       "HEAD," c-rulename.
cs1b:       + termpart.
termpart:   { literal ¦ format }.
a-rule:     a-rulename ":" adef
                 "=>" at1 "=>" at2 "=>" as1 "=>" as2.
adef:       "{" altlist "}".
altlist:    # alt "¦" ... .
alt:        altchar ":" rulename.
at1:        "ALT," a-rulename.
at2:        "{" alt-temp "}".
alt-temp:   # alt-t "¦" ... .
alt-t:      altchar ": NT," rulename.
as1:        "{" a-rulename "}".
as2:        "{" alt-disp "}".
alt-disp:   # alt-d "¦" ... .
alt-d:      altchar ":" rulename.
i-rule:     i-rulename ":" idef
```

```
                "=>" it1 "=>" it2 "=>" is1 "=>" is2.

idef:       "+" rulename1.

it1:        "ITER ( NT," rulename1 "; IOPT," i-rulename ")".

it2:        "NT," rulename1 "; IOPT," i-rulename.

is1:        "$1".

is2:        "[" i-rulename "]".

l-rule:     { l-rule-a
            ! l-rule-b
            ! l-rule-c }.

l-rule-a:   l-rulename ":" ldef-a
              "=>" lt1 "=>" lt2a "=>" ls1 "=>" ls2a "=>" ls3.

ldef-a:     "#" rulename1 rulename2 "...".

lt2a:       "NT," rulename2 "; NT," rulename1
                "; LOPT," l-rulename.

ls2a:       "$1$2".

l-rule-b:   l-rulename ":" ldef-b
              "=>" lt1 "=>" lt2b "=>" ls1 "=>" ls2b "=>" ls3.

ldef-b:     "#" rulename1 "[" rulename2 "]" "...".

lt2b:       "COPT," rulename2 "; NT," rulename1
                "; LOPT," l-rulename.

ls2b:       "$1=[" rulename2 "]$2".

l-rule-c:   l-rulename ":" ldef-c
              "=>" lt1 "=>" lt2c "=>" ls1 "=>" ls2c "=>" ls3.

ldef-c:     "#" rulename1 terminal "...".

lt2c:       "NT," rulename1 "; LOPT," l-rulename.

ls2c:       terminal "$1".
```

```
ltl:        "LIST ( NT," rulenamel "; LOPT," l-rulename ")".

lsl:        "$1".

ls3:        "[" l-rulename "]".

format:     { newline ! tab ! untab }.

newline:    "NL".

tab:        "TB".

untab:      "UT".

extensions: userpdr userpdf.

userpdr:    (undefined) .

userpdf:    (undefined) .
```

APPENDIX E: ILD GRAMMAR LANGUAGE DEFINITION

```
ILD:        langname rulelist [extensions]
    =>  ILD,ILD
          (NT,String;
           NT,rulelist;
           COPT,extensions)
    =>  $1="<langname>" $2 $3="[extensions]" .

rulelist:  + rule
    =>  ITER,rulelist
         (NT,rule;
          IOPT,rulelist)
    =>  NT,rule;
        IOPT,rulelist
    =>  $1
    =>  "[rulelist]" .
```

```
rule:       { c-rule
            ! a-rule
            ! i-rule
            ! l-rule }
     =>  ALT,rule
     =>  { c:NT,c-rule
         ! a:NT,a-rule
         ! i:NT,i-rule
         ! l:NT,l-rule }
     =>  "{rule}"
     =>  "{ c:c-rule ! a:a-rule ! i:i-rule ! l:l-rule }" .

c-rule:     { c-rule-a
            ! c-rule-b }
     =>  ALT,c-rule
     =>  { a:c-rule-a
         ! b:c-rule-b }
     =>  "{c-rule}"
     =>  "{ a:c-rule-a ! b:c-rule-b }" .
```

```
c-rule-a:  c-rulename ":" cdef-a
                  "=>" ctla "=>" csla
    =>  HEAD,c-rule-a
         (NT,String;
          NT,cdef-a;
          NT,ctla;
          NT,csla)
    =>  $1="<c-rulename>" ":" $2 "=>" $3 "=>" $4 .

cdef-a:    + defpart
    =>  ITER,cdef-a
         (NT,defpart;
          IOPT,cdef-a)
    =>  NT,defpart;
        IOPT,cdef-a
    =>  $1
    =>  "[defpart]" .

defpart:   { rulename ¦ option ¦ terminal }
    =>  ALT,defpart
    =>  { r:NT,String
        ¦ o:NT,option
        ¦ t:NT,terminal }
    =>  "{defpart}"
    =>  "{ r:rulename ¦ o:option ¦ t:terminal }" .
```

```
option:     "[" rulename "]"
    =>  HEAD,option
          (NT,String)
    =>  "[" $1="<rulename>" "]" .

ctla:       headop "(" freelist ")"
    =>  HEAD,ctla
          (NT,headop;
           NT,freelist)
    =>  $1 "(" $2 ")" .

headop:     { head | pdf }
    =>  ALT,headop
    =>  { h:NT,head
        | p:NT,pdf}
    =>  "{headop}"
    =>  "{ h:HEAD | p:pdf }" .

head:       "HEAD"
    =>  HEAD,head
    =>  "HEAD" .

pdf:        { (predefined functions) }
    =>  ALT,pdf
    =>  {}
    =>  "{pdf}"
    =>  "{}" .
```

```
freelist:  # freenode ";" ...
    => LIST,freelist
         (NT,freenode;
          LOPT,freelist)
    => NT,freenode;
       LOPT,freelist
    => $1
    => $1
    => "[freenode]" .

freenode:  freeop "," rulename
    => HEAD,freenode
         (NT,freeop;
          NT,String)
    => $1 ";" $2="<rulename>" .

freeop:    { nt ¦ copt }
    => ALT,freeop
    => { n:NT,nt
       ¦ c:NT,copt }
    => "{freeop}"
    => "{ n:NT ¦ c:COPT }" .

nt:        "NT"
    => HEAD,nt
    => "NT" .
```

```
copt:       "COPT"
     =>  HEAD,copt
     =>  "COPT" .

csla:       + dispart
     =>  ITER,csla
          (NT,dispart;
           IOPT,csla)
     =>  NT,dispart;
         IOPT,csla
     =>  $1
     =>  "[dispart]" .

dispart:    { subtree ¦ literal ¦ format }
     =>  ALT,dispart
     =>  { s:NT,subtree
         ¦ l:NT,literal
         ¦ f:NT,format }
     =>  "{dispart}"
     =>  "{ s:subtree ¦ l:literal ¦ f:format }" .

subtree:    "$" integer [opdisfld]
     =>  HEAD,subtree
          (NT,Integer;
           COPT,opdisfld)
     =>  "$" $1="<integer>" $2="[opdisfld]" .
```

```
opdisfld:  { optodf ¦ pdfodf ¦ undodf }
    =>  ALT,opdisfld
    =>  { o:NT,optodf
        ¦ p:NT,pdfodf
        ¦ u:NT,undodf }
    =>  "{opdisfld}"
    =>  "{ o:optodf ¦ p:pdfodf ¦ u:undodf }" .

optodf:    "=""[" rulename "]"""
    =>  HEAD,optodf
         (NT,String)
    =>  "=""[" $1="<rulename>" "]""" .

pdfodf:    "=""<" rulename ">"""
    =>  HEAD,pdfodf
         (NT,String)
    =>  "=""<" $1="<rulename>" ">""" .

undodf:    "=""(" rulename ")"""
    =>  HEAD,undodf
         (NT,String)
    =>  "=""(" $1="<rulename>" ")""" .
```

```
c-rule-b:  c-rulename ":" cdef-b
                "=>" ctlb "=>" cslb
    =>  HEAD,c-rule-b
         (NT,String;
          NT,cdef-b;
          NT,ctlb;
          NT,cslb)
    =>  $1="<c-rulename>" ":" $2 "=>" $3 "=>" $4 .

cdef-b:    terminal
    =>  HEAD,cdef-b
         (NT,terminal)
    =>  $1 .

ctlb:      "HEAD," c-rulename
    =>  HEAD,ctlb
         (NT,String)
    =>  "HEAD," $1="<c-rulename>" .

cslb:      + termpart
    =>  ITER,cslb
         (NT,termpart;
          IOPT,cslb)
    =>  NT,termpart;
        IOPT,cslb
    =>  $1
    =>  "[termpart]" .
```

```
termpart:  { literal ¦ format }
    =>  ALT,termpart
    =>  { l:NT,literal
        ¦ f:NT,format }
    =>  "{termpart}"
    =>  "{ l:literal ¦ f:format }" .

a-rule:    a-rulename ":" adef
                "=>" at1 "=>" at2 "=>" as1 "=>" as2
    =>  HEAD,a-rule
         (NT,String;
          NT,adef;
          NT,at1;
          NT,at2;
          NT,as1;
          NT,as2)
    =>  $1="<a-rulename>" ":" $2
            "=>" $3 "=>" $4 "=>" $5 "=>" $6 .

adef:      "{" altlist "}"
    =>  HEAD,adef
         (NT,altlist)
    =>  "{" altlist "}" .
```

```
altlist:   # alt "|" ...
     =>  LIST,altlist
          (NT,alt;
           LOPT,altlist)
     =>  NT,alt;
         LOPT,altlist
     =>  $1
     =>  $1
     =>  "[altlist]" .

alt:       altchar ":" rulename
     =>  HEAD,alt
          (NT,Character;
           NT,String)
     =>  $1="<altchar>" ":" $2="<rulename>" .

at1:       "ALT," a-rulename
     =>  HEAD,at1
          (NT,String)
     =>  "ALT," $1="<a-rulename>" .

at2:       "{" alt-temp "}"
     =>  HEAD,at2
          (NT,alt-temp)
     =>  "{" $1 "}" .
```

```
alt-temp:  # alt-t ";" ...
    =>  LIST,alt-temp
         (NT,alt-t;
          LOPT,alt-temp)
    =>  NT,alt-t;
        LOPT,alt-temp
    =>  $1
    =>  $1
    =>  "[alt-t]" .

alt-t:     altchar ": NT," rulename
    =>  HEAD,alt-t
         (NT,Character;
          NT,String)
    =>  $1="<altchar>" ": NT," $2="<rulename>" .

as1:       "{" a-rulename "}"
    =>  HEAD,as1
         (NT,String)
    =>  "{" $1="<a-rulename>" "}" .

as2:       "{" alt-disp "}"
    =>  HEAD,as2
         (NT,alt-disp)
    =>  "{" $1 "}" .
```

```
alt-disp:  # alt-d "|" ...
    => LIST,altdisp
         (NT,alt-d;
          LOPT,alt-disp)
    => NT,alt-d;
       LOPT,alt-disp
    => $1
    => $1
    => "[alt-disp]" .

alt-d:     altchar ":" rulename
    => HEAD,alt-d
         (NT,Character;
          NT,String)
    => $1="<altchar>" ":" $2="<rulename>" .

i-rule:    i-rulename ":" idef
                "=>" it1 "=>" it2 "=>" is1 "=>" is2
    => HEAD,i-rule
         (NT,String;
          NT,idef;
          NT,it1;
          NT,it2;
          NT,is1;
          NT,is2)
    => $1="<i-rulename>" ":" $2
          "=>" $3 "=>" $4 "=>" $5 "=>" $6 .
```

```
idef:       "+" rulename1
    =>  HEAD,idef
         (NT,String)
    =>  "+" $1="<rulename1>" .

it1:        "ITER ( NT," rulename1 "; IOPT," i-rulename ")"
    =>  HEAD,it1
         (NT,String;
          NT,String)
    =>  "ITER ( NT," $1="<rulename1>" "; IOPT,"
              $2="<i-rulename>" .

it2:        "NT," rulename1 "; IOPT," i-rulename
    =>  HEAD,it2
         (NT,String;
          NT,String)
    =>  "NT," $1="<rulename1>" "; IOPT," $2="<i-rulename>" .

is1:        "$1"
    =>  HEAD,is1
    =>  "$1" .

is2:        "[" i-rulename "]"
    =>  HEAD,is2
         (NT,String)
    =>  "[" $1="<i-rulename>" "]" .
```

```
l-rule:     { l-rule-a
            ¦ l-rule-b
            ¦ l-rule-c }
    =>  ALT,l-rule
    =>  { a:NT,l-rule-a
        ¦ b:NT,l-rule-b
        ¦ c:NT,l-rule-c }
    =>  "{l-rule}"
    =>  "{ a:l-rule-a ¦ b:l-rule-b ¦ c:l-rule-c }" .

l-rule-a:  l-rulename ":" ldef-a
           "=>"   lt1 "=>" lt2a "=>" ls1 "=>" ls2a "=>" ls3
    =>  HEAD,l-rule-a
         (NT,String;
          NT,ldef-a;
          NT,lt1;
          NT,lt2a;
          NT,ls1;
          NT,ls2a;
          NT,ls3)
    =>  $1="<l-rulename>" ":" $2
            "=>" $3 "=>" $4 "=>" $5 "=>" $6 "=>" $7 .
```

```
ldef-a:    "#" rulename1 rulename2 "..."
    =>  HEAD,ldef-a
         (NT,String;
          NT,String)
    =>  "#" $1="<rulename1>" $2="<rulename2>" "..." .

lt2a:      "NT," rulename2 "; NT," rulename1
               "; LOPT," l-rulename
    =>  HEAD,lt2a
         (NT,String;
          NT,String;
          NT,String)
    =>  "NT," $1="<rulename2>" "; NT," $2="<rulename1>"
             "; LOPT," $3="<l-rulename>" .

ls2a:      "$1$2"
    =>  HEAD,ls2a
    =>  "$1$2" .
```

```
l-rule-b:  l-rulename ":" ldef-b
             "=>" lt1 "=>" lt2b "=>" ls1 "=>" ls2b "=>" ls3
    =>  HEAD,l-rule-b
         (NT,String;
          NT,ldef-b;
          NT,lt1;
          NT,lt2b;
          NT,ls1;
          NT,ls2b;
          NT,ls3)
    =>  $1="<l-rulename>" ":" $2
            "=>" $3 "=>" $4 "=>" $5 "=>" $6 "=>" $7 .

ldef-b:    "#" rulename1 "[" rulename2 "]" "..."
    =>  HEAD,ldef-b
         (NT,String;
          COPT,String)
    =>  "#" $1="<rulename1>" "[" $2="<rulename2>" "]" "..." .

lt2b:      "COPT," rulename2 "; NT," rulename1
               "; LOPT," l-rulename
    =>  HEAD,lt2b
         (COPT,String;
          NT,String;
          NT,String)
    =>  "COPT," $1="<rulename2>" "; NT," $2="<rulename1>"
             "; LOPT," $3="<l-rulename>" .
```

```
ls2b:       "$1=[" rulename2 "]$2"
    =>  HEAD,ls2b
         (NT,String)
    =>  "$1=[" $1="<rulename2>" "]$2" .

l-rule-c:  l-rulename ":" ldef-c
           "=>" lt1 "=>" lt2c "=>" ls1 "=>" ls2c "=>" ls3
    =>  HEAD,l-rule-c
         (NT,String;
          NT,ldef-c;
          NT,lt1;
          NT,lt2c;
          NT,ls1;
          NT,ls2c;
          NT,ls3)
    =>  $1="<l-rulename>" ":" $2
            "=>" $3 "=>" $4 "=>" $5 "=>" $6 "=>" $7 .

ldef-c:     "#" rulename1 terminal "..."
    =>  HEAD,ldef-c
         (NT,String;
          NT,terminal)
    =>  "#" $1="<rulename1>" $2 "..." .
```

```
lt2c:       "NT," rulename1 "; LOPT," l-rulename

    =>  HEAD,lt2c
          (NT,String;
           NT,String)
    =>  "NT," $1="<rulename1>" "LOPT," $2="<l-rulename>" .

ls2c:       terminal "$1"
    =>  HEAD,ls2c
          (NT,terminal)
    =>  $1="<terminal>" "$1" .


lt1:        "LIST ( NT," rulename1 "; LOPT," l-rulename ")"
    =>  HEAD,lt1
          (NT,String;
           NT,String)
    =>  "LIST ( NT," $1="<rulename1>" "; LOPT,"
            $2="<l-rulename>" ")" .

ls1:        "$1"
    =>  HEAD,ls1
    =.  "$1" .

ls3:        "(" l-rulename ")"
    =>  HEAD,ls3
          (l-rulename)
    =>  "(" $1="<l-rulename>" ")" .
```

```
terminal:  literal
    =>  Head,terminal
          (NT,String)
    =>  """" $1="<terminal>" """" .

literal:   literal
    =>  Head,literal
          (NT,String)
    =>  """" $1="<literal>" """" .

format:    { newline ¦ tab ¦ untab }
    =>  ALT,format
    =>  { n:NT,newline
        ¦ t:NT,tab
        ¦ u:NT,untab }
    =>  "(format)"
    =>  "{ n:newline ¦ t:tab ¦ u:untab }" .

newline:   "NL"
    =>  HEAD,newline
    =>  "NL" .

tab:       "TB"
    =>  HEAD,tab
    =>  "TB" .
```

```
untab:      "UT"
    =>  HEAD,untab
    =>  "UT" .

extensions: userpdr userpdf
    =>  HEAD,extensions
          (NT,userpdr;
           NT,userpdf)
    =>  $1 $2 .

userpdr:    (undefined) .

userpdf:    (undefined) .
```

String, Integer, and Character are system predefined rules.

APPENDIX F: MEMORANDUM LANGUAGE DEFINITION

The following Language Definition, constructed by hand, illustrates the templates and schemas required for the definition of a simple grammar. When realized as an AST via the ILD Grammar Directed Editor and interpreted by the system predefined function ILD, this Language Definition could be installed in the Language Definition Module as part of a Memorandum GDE.

```
memo:[salutation] body [closing]
    =>ILD,memo
        (COPT,salutation;
         NT,body;
         COPT,closing)
    =>NL $1="[salutation]" $2 NL TB TB TB $3="[closing]".

salutation:"Dear" name ","
    =>HEAD,salutation
        (NT,String)
    =>"Dear" $1="<name>" "," .

body: + paragraph
    =>ITER,body
        (NT,paragraph;
         IOPT,body)
```

```
    =>NT,paragraph;
      IOPT,body
    =>NL TB UT $1
    =>NL "[paragraph]" .

paragraph: + lines
    =>ITER,paragraph
        (NT,String;
         IOPT,paragraph)
    =>NT,String;
      IOPT,paragraph
    =>$1="<line>" NL
    =>"[line]" NL .

closing:"Sincerely," name
    =>HEAD,closing
        (NT,String)
    =>"Sincerely," NL $1="<name>" .
```

String is a system predefined rule.

APPENDIX G: SYSTEM PREDEFINED FUNCTIONS

The following is a list of programming language primitive operations, derived in part from [Pratt,1975], which could be implemented as System Predefined Functions. This list is not intended as a comprehensive collection of the primitives desired, or even required, for implementation of a GDE system. Rather, these functions are presented here as an indication of the classes of operations which might be made available in support of users of the GDE.

Synthesis Operators

1. NT
2. COPT
3. IOPT
4. LOPT
5. ALT
6. TERM
7. HEAD
8. ITER
9. LIST

Arithmetic Operators

10. PLUS
11. MINUS
12. MUL multiplication

13. DIV division

14. REM remainder

15. UPLUS unary plus

16. UMINUS unary minus

Relational Operators

17. EQUAL equality

18. NTEQ not equal

19. GT greater than

20. LT less than

21. GTE greater than or equal

22. LTE less than or equal

Boolean Operators

23. AND

24. OR

25. NOT

Assignment Operators

26. ASNA arithmetic assignment

27. ASNS string assignment

Sequence Control Operators

28. COND if-then-else conditional

29. LOOP generalized loop

30. CASE

Symbol Table and Data Element Operators

32. DECLARE declaration

33. BLOCK

34. IDENT identifier

35. NUMBER

36. STRING

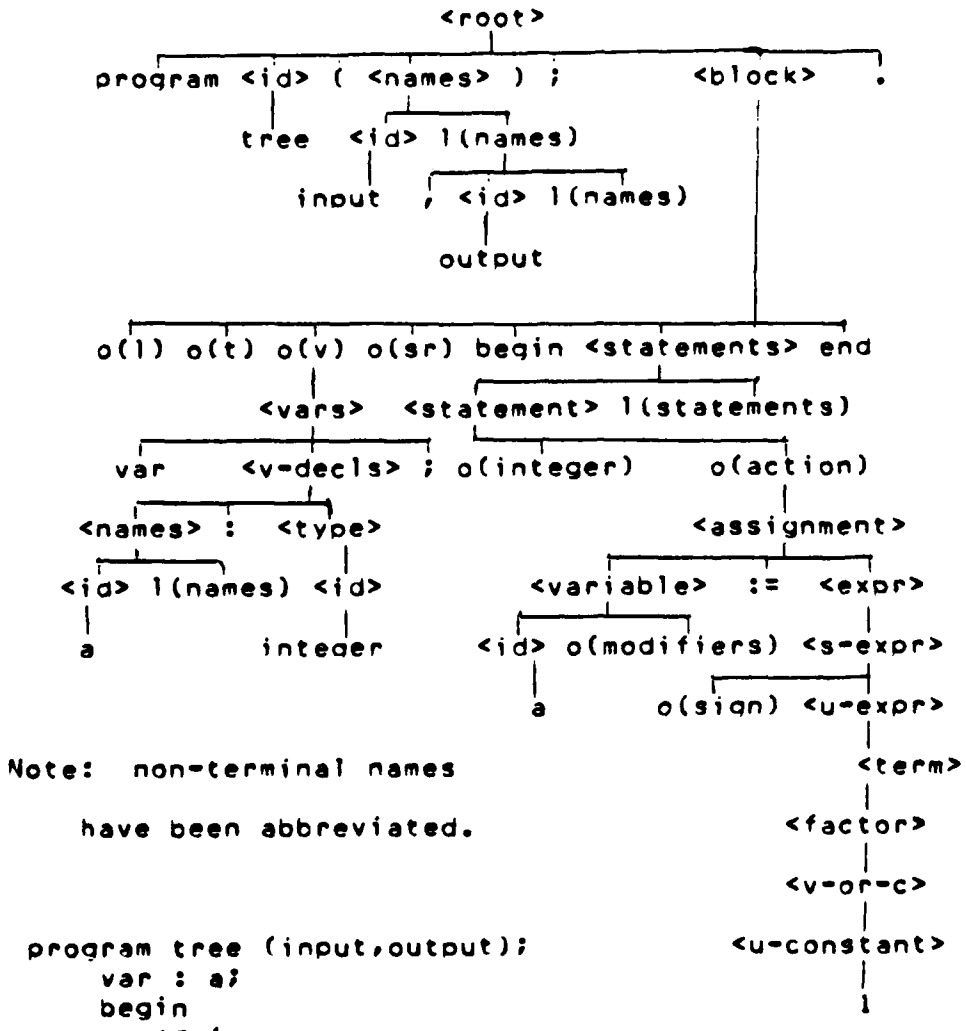
System Operators

37. ILD AST to Language Definition translation

Miscellaneous

38. NOP null operation

APPENDIX H. FIGURES



Note: non-terminal names have been abbreviated.

```
program tree (input,output);
    var : a;
    begin
     a := 1
    end.
```

Figure 1. Parse tree for a trivial program.

```
CONCATENATION:
c : x1 x2 ... xn   ,   xk = { rk | "["rk"]" | tk }

                    <rk>      if xk = rk
    <c>      =>
                    copt(rk) if xk = "["rk"]"

    copt(r) =>  <r>

ALTERNATION:
a : "{" r1 "|" r2 "|" ... "|" rn "}"

    <a>      =>  { <r1> | <r2> | ... | <rn> }

ITERATION:
i : "+" r

    <i>      =>  <r>  iopt(i)

    iopt(i) =>  <r>  iopt(i)

LIST:
l : "#" r1 x "..."   ,   x = { r2 | "["r2"]" | t }

    <l>      =>  <r1>  lopt(l)

                   <r2>  <r1>  lopt(l)       if x = r2
    lopt(l) =>     copt(r2)  <r1>  lopt(l)   if x = "["r2"]"
                   <r1>  lopt(l)             if x = t

PREDEFINED:
p : pdf

    <p>      =>  pdf(p)

UNDEFINED:

    <u>      =>  <u>


        c in C = { concatenation rules }
        a in A = { alternation rules }
        i in I = { iteration rules }
        l in L = { list rules }
        p in P = { predefined rules }
        u in U = { undefined rules }
        r in R = { C,A,I,L,P,U }
        t in T = { terminal symbols }
```

Figure 2. Transformations

```
CONCATENATION:
c : x1 x2 ... xn   ,   xk = { rk ; "["rk"]" ; tk }

                        NT,rk      if xk = rk
    NT,c      =>
                        COPT,rk    if xk = "["rk"]"

    COPT,r    =>  NT,r

ALTERNATION:
a : "(" r1 ";" r2 ";" ... ";" rn ")"

    NT,a      =>  ( NT,r1 ; NT,r2 ; ... ; NT,rn )

ITERATION:
i : "+" r

    NT,i      =>  NT,r  IOPT,i

    IOPT,i    =>  NT,r  IOPT,i

LIST:
l : "#" r1 x "..."   ,   x = { r2 ; "["r2"]" ; t }

    NT,l      =>  NT,r1  LOPT,l

                    NT,r2  NT,r1  LOPT,l     if x = r2
    LOPT,l    =>    COPT,r2  NT,r1  LOPT,l   if x = "["r2"]"
                    NT,r1  LOPT,l            if x = t

PREDEFINED:
p : pdf

    NT,p      =>  PDF(p),p

UNDEFINED:

    NT,u      =>  NT,u

       c in C = { concatenation rules }
       a in A = { alternation rules }
       i in I = { iteration rules }
       l in L = { list rules }
       p in P = { predefined rules }
       u in U = { undefined rules }
       r in R = { C,A,I,L,P,U }
       t in T = { terminal symbols }
```

Figure 3. Labelled Transformations

```
CONCATENATION:
c : x1 x2 ... xn   ,   xk = { rk ; "["rk"]" ; tk }

                    NT,rk      if xk = rk
    NT,c      =>
                    COPT,rk    if xk = "["rk"]"

    COPT,r    =>  NT,r

ALTERNATION:
a : "{" r1 "¦" r2 "¦" ... "¦" rn "}"

    NT,a      =>  ALT,a
    ALT,a     =>  { NT,r1 ; NT,r2 ; ... ; NT,rn }

ITERATION:
i : "+" r

    NT,i      =>  NT,r  IOPT,i
    IOPT,i    =>  NT,r  IOPT,i

LIST:
l : "#" r1 x "..."   ,   x = { r2 ; "["r2"]" ; t }

    NT,l      =>  NT,r1  LOPT,l

                    NT,r2  NT,r1  LOPT,l    if x = r2
    LOPT,l    =>    COPT,r2  NT,r1  LOPT,l  if x = "["r2"]"
                    NT,r1  LOPT,l           if x = t

PREDEFINED:
p : pdf

    NT,p      =>  TERM,p
    TERM,p    =>  PDF(p),p

UNDEFINED:

    NT,u      =>  NT,u

        c in C = { concatenation rules }
        a in A = { alternation rules }
        i in I = { iteration rules }
        l in L = { list rules }
        p in P = { predefined rules }
        u in U = { undefined rules }
        r in R = { C,A,I,L,P,U }
        t in T = { terminal symbols }
```

Figure 4. Extended Transformations

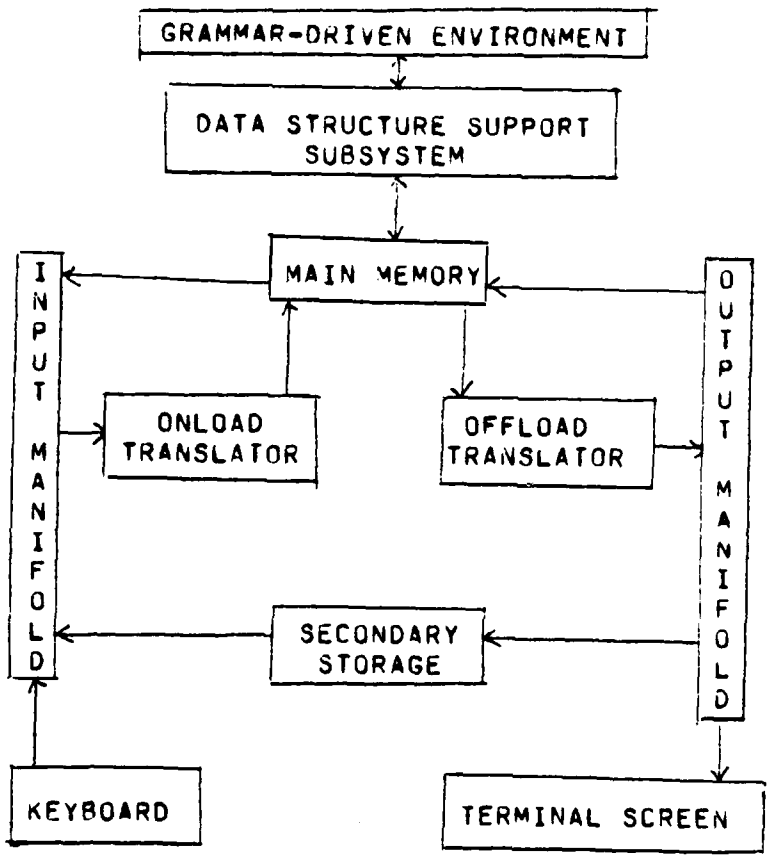


Figure 5. System Architecture (Data Flow)

LIST OF REFERENCES

Aho, A.V. and Ullman, J.D., Principles of Compiler Design, p. 211-224, Addison-Wesley, 1977.

Fraser, C.A., "A Generalized Text Editor", Communications of the Association for Computing Machinery, v. 23, p. 154-158, March 1980.

Habermann, A.N., "An Overview of the Gandalf Project", Carnegie-Mellon University Computer Science Research Review 1978-79, 1979.

Hopcroft, J.E. and Ullmann, J.D., Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, 1979.

Iverson, K.E., A Programming Language, John Wiley, 1962.

Kroenke, D., Database Processing, Science Research Associates, 1977.

MacLennan, B.J., Semantic and Syntactic Specification and Extension of Languages, Ph.D. thesis, Purdue University, 1974.

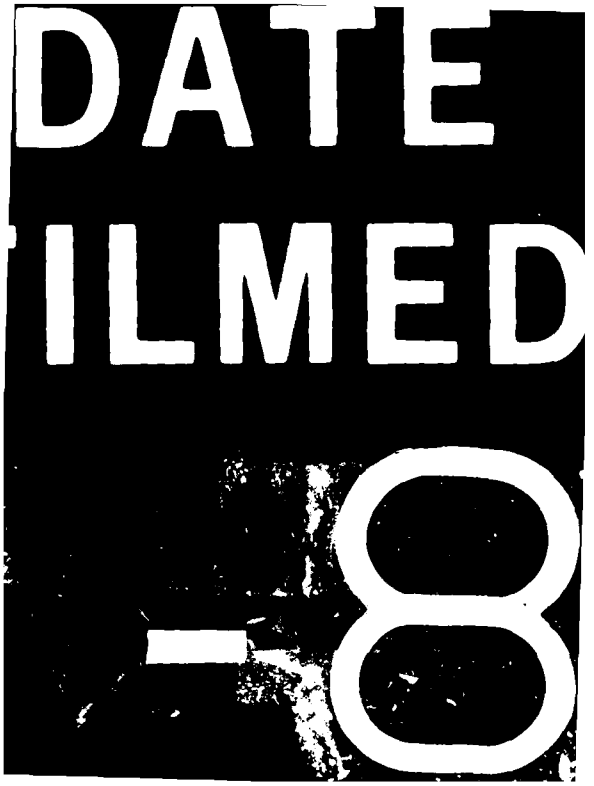
McKeenan, W.M., "Compiler Construction" in Compiler Construction: An Advanced Course, F.L. Bauer and J. Eickel, ed., Springer-Verlag, 1976.

Pratt, T.W., Programming Languages: Design and Implementation, Prentice-Hall, 1975.

Center for Research in Computing Technology, Harvard University, Technical Report 23-74, FLL Programmer's Manual, by Wegbreit, B., et. al., 1974.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	5
4.	Asst. Prof. Bruce J. MacLennan, Code 52Ml Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
5.	Asst. Prof. Douglas R. Smith, Code 52Sc Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
6.	LCDR William R. Shockley, Code 52Sp Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
7.	LT Daniel P. Haddow, Code 52Hw Department of Computer Science Naval Postgraduate School Monterey, California 93940	2


DATE
ILMED
8