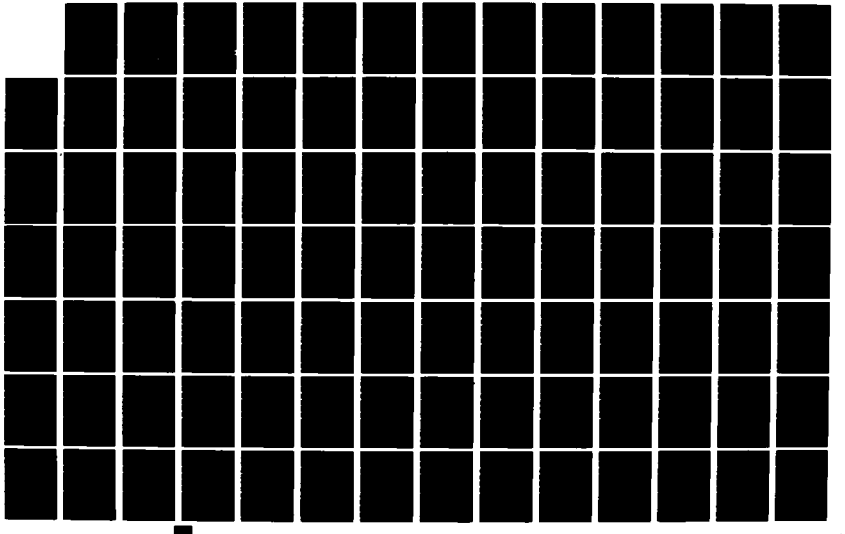
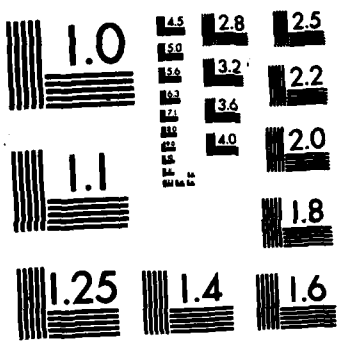
AD-A124 769 ADAPAR: AN ADA RECOGNIZER(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGINEERING W R URE DEC 82 AFIT/GCS/MA/82D-11 1/2

UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD A124139

DTIC FILE COPY

This document has been approved for public release and sale; its distribution is unlimited.

DTIC
ELECTE
FEB 23 1983
S D
A

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY (ATC)

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/MA/82D-11

ADAPAR: AN ADA RECOGNIZER

THESIS

AFIT/GCS/MA/82D-11 William R. Ure
2Lt USAF

DTIC
ELECTE
FEB 2 3 1983
A

Approved for public release; distribution unlimited.

AFIT/GCS/MA/82D-11

ADAPAR:

AN ADA RECOGNIZER

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

by

William R. Ure
2Lt USAF

Graduate Computer Science

December 1982

DTIC COPY INSPECTED 2

Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A

Approved for public release; distribution unlimited.

PREFACE

My original interest in computers was piqued in elementary school. The school received a grant from the National Science Foundation to timeshare on a computer. This interest continued through secondary school and into college. It was only natural that I pursue computer science in college.

My interest in compilers and formal languages was motivated by the numerous programming language courses I attended. I also took a class in compilers while a undergraduate to learn about the inner workings of a compiler. Unfortunately. this was just one seven week course and there was insufficient time to delve into very much detail.

At the Air Force Institute of Technology, I availed myself of the opportunity to take the three course compiler sequence. The courses were capably taught by Capt. Roie R. Black, who ultimately became my thesis advisor.

The original focus of this thesis effort was to replace the table driver parser in the AFIT-Ada compiler with a recursive descent parser, as well as add additional

capabilities to it.

As I began to implement the parser, it became apparent that programming the procedures for the parser was going to occupy much of time allotted for this effort.

The result of this thesis is an Ada recognizer. The Ada recognizer will determine whether or not a given Ada program is syntactically correct. The Ada recognizer is a tool that has a variety of potential uses.

I would like to thank Major Israel Caro and Lt. Dan Ehrenfried and the Avionics Laboratory for their support and the use the AVSAIL DecSystem-10. A majority of the software development was done on this computer. I would also like to thank the people of SYSTRAN Corporation and Digital Equipment Corporation for their aid and support. Additionally, I would like to thank Mark Pfoff, the night operator, for putting up with me late at night.

I would like to thank my advisor Capt. Roie Black for his advice, patience, understanding and support, particularly when this effort seemed bogged down.

Table of Contents

PREFACE	ii
ABSTRACT	vii
1. Introduction	1
1.1 Objective	1
1.2 Background	1
1.3 Assumptions	3
1.4 Overview	3
2. COMPILERS	5
2.1 COMPILERS	5
2.2 The Compilation Process	6
2.2.1 Lexical Analysis	7
2.2.2 Syntax Analysis	9
2.2.3 Semantic Analysis	12
2.2.4 Code Generation	13
2.2.5 Interpretation / Loading	14
2.3 Language Structure	15
2.3.1 Grammars	16
2.3.2 Classes of Grammars	18
2.4 FIRST sets	20
3. ADA Recognizer	22
3.1 Origin	22
3.2 Design Approach	23
3.3 Syntax Diagrams	24
3.3.1 Ada Syntax Diagrams	27
3.4 Lexical Analyzer	30
3.5 Symbol Table	33
3.5.1 Name Node	35
3.5.2 Information Node	37
3.5.3 Other Symbol Table Issues	37
3.6 The Parser	37
3.7 Example	43
3.8 Other Issues	50
3.8.1 Error Recovery	50
3.8.2 Software Engineering Concerns	50
4. Evaluation	52
4.1 Syntax Directed Editor	52

4.2 Syntax Verification 53

5. Recognizer to Compiler Transformation 55

5.1 Symbol Table 55
5.1.1 Symbol Table Management 56
5.2 Semantic Analysis 60
5.3 Code Generation 61

6. Conclusions 63

6.1 Current Status 63
6.2 Recommendations 64
6.3 Conclusion 65

BIBLIOGRAPHY 67

A. Bonet Syntax Diagrams 68

VITA 87

List of Figures

Figure 2-1:	Example Ada Statement	7
Figure 2-2:	Fortran Example	8
Figure 2-3:	Example Statement	12
Figure 2-4:	Example Production Rule	16
Figure 2-5:	context free production	18
Figure 2-6:	A context Free Grammar	19
Figure 2-7:	Use of Alternation Symbol	19
Figure 3-1:	Left Recursive Diagram	28
Figure 3-2:	Lexical Analysis Structure	32
Figure 3-3:	Structure of Hash Node	34
Figure 3-4:	Name Storage Structure	36
Figure 3-5:	BNF for TYPE_DECLARATION	43
Figure 3-6:	BNF Associated with TYPE_DEFINITION	44
Figure 3-7:	TYPE_DEFN Diagram	46
Figure 3-8:	TYPE_DECL Diagram	47
Figure 3-9:	Procedure type_decl	47
Figure 3-10:	Procedure type_defn	49
Figure 5-1:	Example Declaration	56
Figure 5-2:	Parser Code for Declaration	56
Figure 5-3:	Symbol Table Code Added	58

ABSTRACT

This thesis involved the development of a top down recursive descent Ada recognizer. Basic concepts of compiler theory as they relate to syntax analysis were reviewed. Appropriate syntax diagrams were selected and transformed into program statements using a structured method. The software was developed with attention to software engineering practices. Uses for the recognizer as a programmers tool are discussed. The steps necessary to transform the recognizer into a compiler are discussed. The development of the Ada recognizer was performed on the DecSystem-10 of the Air Force Avionics Laboratory at Wright-Patterson AFB, Ohio.

1. Introduction

1.1 Objective

The objective of this thesis effort is to implement an Ada recognizer. The recognizer accepts syntactically valid Ada programs and rejects those that are not. Originally, the goal was to build a recursive descent parser for Ada and use it to replace the table driver parser in the AFIT-Ada compiler developed by Garlington and Werner [Ref 5, 12]. It soon became evident that coding the recursive descent parser would take more effort than was originally assumed. Therefore, the revised objectives of this thesis are to review basic compiler concepts, and to provide a flexible tool that others might be able to use in future efforts aimed at developing a full Ada language capability at AFIT.

1.2 Background

The Department of Defense's (DoD) primary reason for developing the programming language Ada was to reduce software costs. A recent study concluded that the DoD was spending in excess of three billion dollars ($3,000,000,000) on embedded computer software [Ref 12]. This trend was expected to increase as computer became an integral part of defense systems. A large part of this cost was due to the plethora of language being used for defense systems software.

Some of these languages were specially developed for one particular application and therefore not very useful for more general applications. In these cases, the cost of software development included not only the applications programs, but the implementation and validation of a compiler for the new language.

Another problem with the proliferation of languages was the amount of personnel resources that were consumed. Personnel working on a given project that used one of these custom languages had to be trained to use the language. Personnel could not readily be transferred from one project to another without extensive retraining. This also contributed to the escalating cost of software.

The proposed solution to the problem was to design one language that could be used for any DoD embedded computer application. This language would need many features required by various systems. These included tasking, capability to implement realtime software and program controlled exception handling.

The resulting language was designed and called DoD-1. Later is was renamed Ada in honor of Lady Augusta Ada Byron, Countess Lovelace, who is acknowledged to be the world's first computer programmer.

1.3 Assumptions

It was assumed that a top down recursive descent parser was a legitimate approach to implementing the parser, since compilers for other languages such as Pascal and Algol-60 are commonly implemented that way. Ada bears a considerable resemblance to these languages. In addition, it was assumed published syntax diagrams developed by Bonet [Ref 3] for Ada were correct. It was discovered that the diagram for **TYPE_DECLARATION** was missing. A diagram for **TYPE_DECLARATION** was derived from the grammar in the Ada Reference Manual [Ref 11] using Wirth's method [Ref 13]. This new diagram is developed in section 3.7. The Bonet diagrams are included in the appendix for reference.

It was assumed that the results of previous efforts were available for use on the Avionics Lab DecSystem-10.

1.4 Overview

The Ada recognizer determines if a given Ada program is well formed or not. It does not generate any run time code. To achieve this, the lexical and syntactical analysis portions of an Ada compiler were implemented. This implementation recognizes the entire language set for Ada. The method used is a top down recursive descent parser. The Ada recognizer is a necessary first step toward a full compiler and has several other uses as well and is therefore

a topic of interest to the Air Force.

Before the Ada recognizer is discussed, it is necessary to present some of the basic theories of compilers. These are discussed in Chapter 2. The development and implementation of the Ada recognizer is discussed in Chapter 3. Chapter 4 presents the evaluation and possible uses of the Ada recognizer.

The original goal of the project was to incorporate the recursive descent parser into the AFIT-Ada compiler. Chapter 5 explains the steps necessary to transform the Ada recognizer into a compiler.

The current status of the software developed for this effort as well as conclusions and recommendations are presented in Chapter 6.

The appendix contains the Bonet syntax diagrams.

2. COMPILERS

2.1 COMPILERS

There are two basic ways to implement higher-level languages. These involve building translators and compilers. A translator takes as input a program written in a high level language, the source language, and transforms it into another high level language, called the object language. Compiler transform the source language into a low-level language, such as assembler or machine code.

Once upon a time, compilers were considered almost impossible programs to write. For example, the first Fortran compilers took 18 man-years to implement. Today, compilers can be implemented with much less effort. [Ref 1:1]

Some of the reasons for this progress include:

- A new understanding of how to organize and modularize the compilation process
- Software tools that automate the more mundane tasks of compiler construction
- Systematic techniques for handling many tasks performed by the compiler
- Improved language design and specification, due to intense study of programming languages as formal languages

The last point will be pursued further when grammars are discussed.

The compilers of today are better organized, more understandable and in general, more efficiently implemented than those of 20 years ago.

2.2 The Compilation Process

Traditionally, the compilation process is divided into five subprocesses. These are:

- lexical analysis
- syntactic analysis
- semantic analysis
- code generation
- interpretation / loading

These operations are usually performed in the order indicated. This does not necessarily mean that five passes through the source program are necessary. Modern compiler techniques allow for one pass compilation, provided the language meets certain criteria. In this case, the above operations are done in parallel.

Let us now look at each of these subprocesses in more detail.

2.2.1 Lexical Analysis

The lexical analysis portion of the compiler can be thought of as a filter. It effectively removes items from the source file that are not needed by other parts of the compiler. Such items include extraneous spaces, format control characters and comments. The lexical analyzer is also responsible for reporting any illegal characters it may find in the source program.

The lexical analyzer takes as input the text from the source program. The output is a stream of symbols that are meaningful to the syntax analyzer. These symbols are called tokens. A token is the atomic unit that the compiler deals with. Each token represents a sequence of characters that can be treated as a single logical entity. Reserved words, identifiers, punctuation and numbers are typical tokens.

For example the ADA statement:

```
IF COUNT = 5 THEN NEWPAGE;
```

Figure 2-1: Example Ada Statement

contains seven tokens: IF, COUNT, "=", 5, THEN, NEWPAGE, and ";". It is not feasible to consider each letter of an identifier as a token, since the letters by themselves do not

represent any particular entity.

The string of characters designated as a token depends largely on the source language and the design of the compiler writer.

In Ada, there are several sequences of symbols that are treated as tokens, even though subsequences of these may also be tokens. Examples of this are: "<<", ">>", "**", and "/=". "<" can be a token by itself, but when followed immediately by another "<", the two characters together take on a different meaning.

In order to locate a token, the lexical analyzer scans the input source character by character beginning from some initial character position, examining successive characters until it finds a character that may not be logically grouped with the previous characters to form a token. In some languages, this may require reading several characters after the actual end of token to obtain the correct token. A classic example from Fortran illustrates this.

```
DO10I=1,5
```

Figure 2-2: Fortran Example

The lexical analyzer would need to read to the comma to

determine that the next token should be the keyword DO. Fortunately, the ADA lexicon is designed so that lexical analyzer would need to look ahead at most one character.

In many cases, lexical analysis can be performed in parallel with syntax analysis. The syntax analysis routine can simple call lexical routines to deliver the next token to it.

2.2.2 Syntax Analysis

The second phase of the compilation process is syntax analysis, or parsing. Parsing insures that the source program is well-formed. That is, it conforms with the specification of the syntax of the language. The parser examines the stream of tokens to insure compliance with the syntax specification.

The method of parsing can take many forms. Parsing has been the subject of intense study. It falls into the category of systematic techniques for handling compiler tasks.

A very common method of parsing is known as bottom up parsing. This was the method used by Garlington and Werner in their work on the AFIT-ADA compiler [Ref 5,12]. The basic bottom up method consists of building a finite state machine for the language and working through it, using the tokens to

determine which transitions to take. This is tantamount to building a graph. The vertices of the graph are the states of the parser and the edges are labeled with the tokens that indicate the transition. In a given state, the parser examines the current token. If there is an edge of the graph out of that state labeled with the token, the parser moves to the state indicated by that edge. If there is no transition for that token, the parser must back up and attempt to find another valid path through the graph or else report an error.

For a language such as ADA, building the finite state machine is a very tedious process, There would be several hundred states and even more transitions. Fortunately, there are automated tools available to build the finite state machine. These tools take the source language specification as input, and output the tables needed for the parser.

There are several disadvantages to this method. It is very difficult to determine exactly what is taking place during parsing. It is cumbersome to work through the tables by hand to locate an error. It is also difficult to look at the program and get some idea of what it is doing and what state it is in. It must be absolutely certain that the specification to the automated tool that generates the tables is correct, otherwise the resulting tables will contain errors.

One advantage to this method is that the parser is relatively easy to implement, once the tables are verified as correct. The procedures needed to do the parsing would simply examine the token stream and follow the tables. Errors are reported when there is no valid path through the graph for the given input.

Another popular method is known as the top down recursive descent parser. This is the method employed in this effort. A recursive descent parser is a collection of mutually recursive procedures. Each procedure is charged with recognizing a specific construct of the language syntax. The chief tool used in constructing a recursive descent parser is a set of syntax diagrams for the language. Currently, there are several sets of diagrams available for ADA [Ref 3, 4].

There are several advantages to this method of parsing. Since each routine is responsible for recognizing a specific construct in the language, it is easy to follow the diagrams along with the parser to determine what state the parser is in. This makes debugging easier. After the parser is built and verified, semantic routines and code generation modules can be installed in the parser routines.

There are several disadvantages to this method. Implementing the parser takes a considerable coding effort,

particularly in a large language such as ADA. Another problem is that many of the constructs get quite complicated, requiring attention to the nesting of control constructs. The direct approach and readability of this method makes some of the disadvantages more palatable.

2.2.3 Semantic Analysis

Semantic analysis determines exactly what the program statements mean and whether or not they make sense. Syntax analysis insures that the source program conforms to the grammar of the language, while semantic analysis insures that the program statements are meaningful. Consider the following:

```
A := B + C;
```

Figure 2-3: Example Statement

This statement is syntactically correct, but it may not be semantically correct. It depends on what A, B and C represent. If A represents an integer, B a boolean value and C an array of characters, this statement does not make any sense. However, if A, B, and C are real numbers, then it is obvious what this statement means and it does make sense.

Semantic analysis not only determines the type of variables and results, but also determines the actions of certain operators. In the above example, '+' may be interpreted several different ways depending on the types of A,B and C. If A,B and C are integers, '+' represents addition. If A,B and C are boolean , then '+' represents the logical or function in Ada. '+' may also be defined as a special function for a specific type of A,B and C.

At present, there are no specific tools available to specify the semantics of a language, although developing them is receiving attention in academic circles [Ref 10]. Usually, language semantics are defined by written descriptions.

In general, the complexity of language semantics depend on how strongly typed the language is. Fortran has very few specific semantic restrictions, while ADA has very complex semantics. Fortran is weakly typed, while Ada is strongly typed.

2.2.4 Code Generation

Code generation is the primary reason for the existence of compilers. Lexical, syntactical and semantic analysis are needed to insure that the code that is generated will do what it is supposed to. The goal of the compiler is to produce machine code that will accomplish the actions implied by the

statements in the source code for some actual computing machine.

Code generation can take many forms. The crudest method is to generate object code directly from the output of the semantic analyzer. While this will produce object code that is correct, it will be inefficient. A more elegant method is to produce some sort of intermediate code. The intermediate code can then be processed by an optimizer to produce more efficient object code.

The price paid for this efficient code is an increase in compilation time. This may be acceptable if the object code is going to be used frequently. A little extra compilation time is better than having a slow production program.

2.2.5 Interpretation / Loading

After code generation is complete, the object code remains. Something must be done with it to make it useful. If the object code is for the machine that the source was compiled on, then we could just load it into memory, along with any associated modules such as library routines, standard packages or other compilation units.

If the object code produced is for another machine, it must be loaded on that machine or an interpreter can be used to run the object code. Basically, the interpreter simulates

the intended target machine. This technique is used for testing software where testing on the actual machine is impractical or dangerous, as would be the case in a satellite or missile system.

Now that the basics of compilers have been presented, some aspects of languages structure are presented. This is appropriate since language structure is an important part of compilers, particularly syntax analysis.

2.3 Language Structure

The elements of a language are its alphabet, grammar, and semantics. These elements correspond to lexical analysis, syntactical analysis, and semantic analysis, respectively. The term sentence is used to refer to a finite string of symbols in the alphabet. These symbols are called tokens in the context of compilers. Tokens should not be confused with individual elements of the graphic character set. For example, INTEGER is composed of seven graphical characters. Individually, these letters have no special meaning by themselves. When they are concatenated together, they take on meaning and become a symbol of the language alphabet.

Similarly, when dealing with the English language, each letter is not examined individually to determine the meaning of a sentence. It is the words and their positional

relationship that give meaning to the sentence. Words are the tokens of the English language.

2.3.1 Grammars

A grammar is used to formally specify the syntax of a formal language. A grammar is specified by the 4-tuple: (N, T, P, S), where N is a set of special symbols called non-terminal symbols, T is the set of symbols representing the alphabet, called terminal symbols, P is a set of production or rewriting rules and S is a special non-terminal symbol called the start symbol or goal symbol.

A non-terminal is a symbol that can represent a string of terminal and other non-terminal symbols. A terminal symbol is any member of the alphabet. A token and a terminal symbol are essentially the same thing. However, terminal is generally used in connection with grammars and token is used when discussing other aspects of compiler implementation. Note that non-terminals are not part of the alphabet.

A production rule, or rewrite rule has the form:

x ===> y

Figure 2-4: Example Production Rule

where x and y are members of the set of strings composed of terminal and non-terminal symbols,(N U T)*. Restrictions on the exact composition of x and y depend on the type of grammar being examined.

Productions are applied by replacing x with y in a string. Thus, the string 'axb' would become 'ayb' after applying the production rule.

If we start with any string at all, we could theoretically obtain any string of the set (N U T)*. This is obviously not the intent of using a grammar. Instead, a small set of strings are chosen as starting strings. Generally one non-terminal, S, is used to represent the starting strings.

The starting symbol, S, is replaced by one of the choices from its production set. If the resulting string

contains non-terminal symbols, they are replaced by choices from their corresponding production set. This process is iterated until there are only terminal symbols left in the string.

2.3.2 Classes of Grammars

There are three basic types of grammars used in language theory: context sensitive, context free, and linear. These grammars are called phrase structured grammars since the productions are expressed in sentential form. Grammars that are not phrase structures are called unrestricted. Unrestricted grammars are of little use for compiler implementation.

Context sensitive grammars and linear grammars are not normally used in describing computer languages and are not applicable to this effort. Therefore, they will not be discussed here.

Context free grammars are most commonly used to specify the syntax of a programming language. Productions of a context free grammar are of the form:

x ===> y

Figure 2-5: context free production

where x is a member of N and y is a member of (N U T)*. Note that x represents one and only one non-terminal. An example of a context free grammar is an expression grammar that preserves the precedence of operators.

G = (N, T, S, P)

where N = {E, T, F}

T = {+, (,), *, a}

S = E

and P = {

E ===> E + T
E ===> T
T ===> T * F
T ===> F
F ===> (E)
F ===> a
}

Figure 2-6: A context Free Grammar

Occasionally, there will be two or more productions for a given non-terminal. These can be combined into a single production using the alternation symbol (|). Thus the first two productions in the example can be combined to form the following production rule.

E ===> E + T | T

Figure 2-7: Use of Alternation Symbol

The context free grammar is a sufficiently powerful tool to specify the syntax of most programming languages. Furthermore, this class of grammar has been studied extensively, so there are systematic methods of producing a parser from a given context free grammar. The method used in this effort will be discussed in the next chapter.

2.4 FIRST sets

One other element of formal language theory must be presented, because it is essential to the implementation of a recursive descent parser. This is the theory of FIRST sets. A FIRST set is merely the set of initial terminal symbols for a given non-terminal. Consider the example grammar from figure 2-6. The FIRST set of the non-terminal F is {(, a}. Occasionally it may be necessary to compute the FIRST set of another non-terminal to derive the FIRST set of a particular non-terminal For example, the FIRST set of E, usually written FIRST(E), is {E, T}. FIRST(T) must now be computed. FIRST(T) = {T, F}. Now, FIRST(F) must be computed. As stated previously FIRST(F) = {(, a}. Since the elements of the set are all terminal symbols, the process is complete. Note that FIRST(E) is not computed since this would erely

duplicate entries in the final result.

Now that the necessary aspects of formal language theory have been presented, the development of the Ada recognizer can be discussed.

3. ADA Recognizer

In this chapter, the process followed to develop the Ada recognizer is discussed. The Ada recognizer is basically a top-down recursive descent parser with two symbol lookahead. The parser was designed to recognize the full Ada language.

Before discussing the specifics of how the recognizer was constructed, a brief presentation of the origin is appropriate.

3.1 Origin

The origin of the Ada recognizer can be traced to many sources. The original motivation for this work was the AFIT-ADA compiler by Garlington [Ref 5], revised by Werner [Ref 12]. The basic idea was to eventually incorporate the Ada recognizer into the AFIT-ADA compiler, effectively replacing the original table driver parser with a recursive descent parser.

An additional source for this project was the revised Ada language reference manual [Ref 11]. The revised manual more precisely specified many aspects of the language than the original version. In addition it cleared up many ambiguities that the original manual contained.

Wirth's PL/0 compiler [Ref 13] provided the basis on

which the Ada recognizer was built. The parser in the PL/0 compilers is a recursive descent parser. Wirth has also devised a systematic approach for building a recursive descent parser from a set of syntax diagrams.

3.2 Design Approach

The design approach used to construct the parser was top-down analysis. Therefore, the goal, and Ada recognizer, is the appropriate starting point. It was decided that constructing a parser for the entire language was not unreasonable. The next issue was how the parser should be structured. A recursive descent parser was used, because languages such as Pascal and Algol-60 are implemented in this manner. These are both procedure oriented languages and their syntax bears a considerable resemblance to Ada. Pascal was selected as the implementation language for the parser because it would be relatively simple to translate the Pascal parser into Ada when a suitable compiler became available.

Now that the design method has been selected, some formal specification of the Ada syntax would be needed. Several different forms are available. The formal specifications of the syntax fall into two general categories. These are Backus Normal Form (BNF) type specifications and syntax diagrams. Syntax diagrams are normally generated from a BNF grammar definition. The syntax

diagrams themselves are more appropriate for a recursive descent parser implementation because they are basically a flowchart for the parser. The selection of appropriate syntax diagrams is discussed in section 3.3.

The implementation of lexical analysis is presented in section 3.4. The uses of the symbol table during syntax analysis are described in section 3.5. Using the above mention material as a foundation, the construction of the parser can be presented. This is discussed in section 3.6. An example of the transition from a BNF type grammar to Pascal procedures is presented in section 3.7. Other issues that are important are discussed in section 3.8. These include error recovery and software engineering.

3.3 Syntax Diagrams

The key to building a recursive descent parser is the availability of a suitable set of syntax diagrams. Wirth [Ref 13] presents a structured method for producing syntax diagrams from a context free grammar.

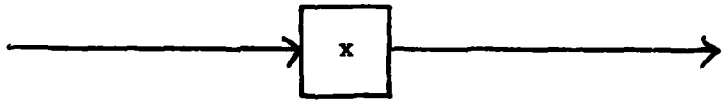
Rules of Graph Construction:

G1. Each nonterminal symbol, A, with corresponding production set:

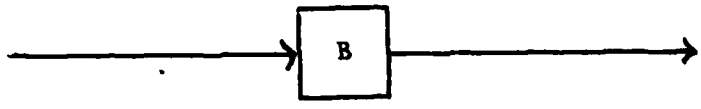
A ::= E1 | E2 | E3 | ... | En

is mapped into a syntax graph, A, whose structure is determined by the right hand side of the production according to Rules G2 through G6.

G2. Every occurance of a terminal symbol x in Ei corresponds to a syntax statement for this symbol and the removal of this symbol from the input stream. This is represented by the following graph:



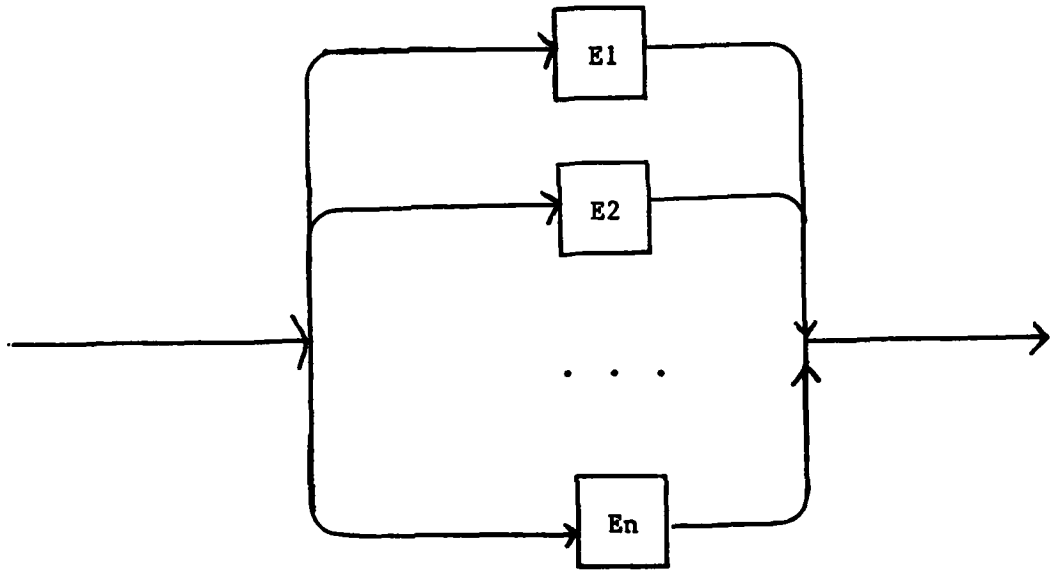
G3. Every occurance of a non-terminal symbol, B, in a Ei corresponds to an activation by syntax graph B. This is represented by the following:



G4. A production having the form

A ::= E1 | ... | En

is mapped into the graph

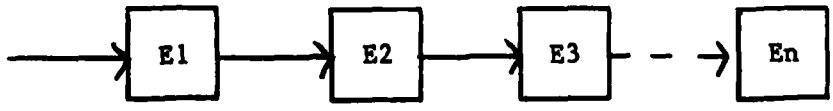


where each Ei in a box is obtained by applying construction rules G2 through G6 to Ei.

G5. An E having the form

E = E1 E2 E3 ... En

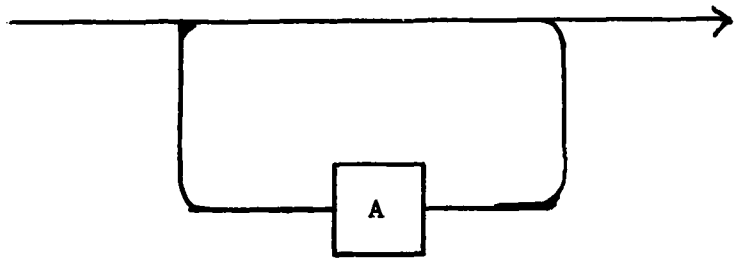
is mapped into the graph



G6. An E having the form

E = { A } where { A } means zero or more occurances of A

is mapped into the graph



where the A in the box is obtained by applying construction rules G2 through G6 to A.

Once all of the graphs have been derived, they can be folded into one another. This process is examined in the example in section 3.7. This seems to be the general process that Bonet used to generate the syntax diagrams used in this effort.

3.3.1 Ada Syntax Diagrams

Fortunately, syntax diagrams for the Ada language exist, so it was not necessary to perform the preceding process for the entire language. Two sets of diagrams were considered for this effort. These were due to Bonet et al [Ref 3] and DeRemer et al [Ref 4]. Both sets of diagrams contained

deficiencies. The Bonet diagrams were selected over the DeRemer diagrams for the following reasons.

The Bonet diagrams were verified to be of a form necessary or a recursive descent parser to work correctly. Basically, this means that there are no left recursive diagrams in the set. An example of a left recursive diagram is shown in figure 3-1.

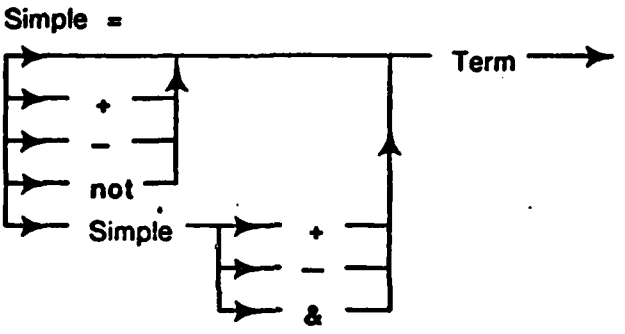


Figure 3-1: Left Recursive Diagram

Notice that by following the diagram SIMPLE, there is a path that leads right back to SIMPLE. This is left recursion. Left recursion is unacceptable because it may cause the parser to go into a recursively infinite loop. That is, when trying to interpret SIMPLE the parser may attempt to interpret SIMPLE again without processing any input symbols. This cycling will almost always occur when an erroneous construct is encountered. It may also occur on a

legal input string if the decision to use SIMPLE again is not the last resort. In the example, if the path for SIMPLE is considered before the path for +, -, or **not**, the parser may go into an infinite loop.

The preceding example comes directly from the DeRemer diagrams. This was the primary reason for not using them. It would have been tedious and non-productive to correct all of these cases.

Another reason that the DeRemer diagram were rejected is that they are generally larger than the Bonet diagrams. Since the Bonet diagrams are smaller, their resulting implementation is more modular. The implementation of large diagrams is complicated by restrictions in the DecSystem-10 Pascal compiler.

The Bonet diagrams do have some deficiencies. The major problem is that they require two symbol lookahead in four places. This means that the parser would need to look ahead two symbols in these places in order to determine which path to follow. The DeRemer diagrams only require a one token look ahead. This problem is easily solved by having the lexical analyzer deliver the next two tokens to the syntax analyzer. The mechanics of this are discussed in the next section.

One other problem with the Bonet diagrams was that the diagram for **type_declaration** was omitted. This diagram was generated from the Ada reference grammar using Wirth's method, presented earlier.

The deficiencies of the Bonet diagrams were much less sever than those of the DeRemer diagrams, therefore the Bonet diagrams were selected for implementation.

3.4 Lexical Analyzer

Before attempting to implement syntax analysis, it is necessary to implement lexical analysis, because the syntax analyzer must use the output of the lexical analyzer. The specifics of lexical analysis are discussed in this section.

The lexical analysis for Ada is straightforward. If the first character of a token is a digit, the token is a number. If the first character is alphabetic, the token is either a reserved word or an identifier. If the first character is any other character from the legal character set, the token is a delimiter token. Format effector characters such as tabs, spaces and carriage returns are treated as delimiters, but are not considered tokens. Ada requires that tokens be delimiters or be separated by delimiters, so that the problems encountered with the Fortran "DO" statement do not exist. (See figure 2-2).

The delimiter characters are divided into two categories. The first category is those characters that cannot be part of a two character token. Examples include "(", ")", and ";". The second category is characters that can start a two character token. These are called prefix characters. For example, "<" may be part of "<=", "<<", or "<>" or may stand by itself. Once the token is determined, it is assigned a token value through the use of a Pascal enumeration type.

The main lexical routine is called **GETTOK**. **GETTOK** is responsible for returning the next two tokens in the input stream. It uses some subordinate routines that have specific functions.

GETALPHANUM is responsible for getting an alphanumeric token such as a reserved word or an identifier. This routines also builds the linked list to store the lexical value of the token. This will be discussed further in the next section.

GETNUMBER returns a numeric token and its value in base 10.

GETCH returns the next two character in the input stream. It also takes care of extraneous things like eliminating blank lines. This routine is used by the three

previously mention procedures.

The following diagram best illustrates the structure of the lexical analyzer.

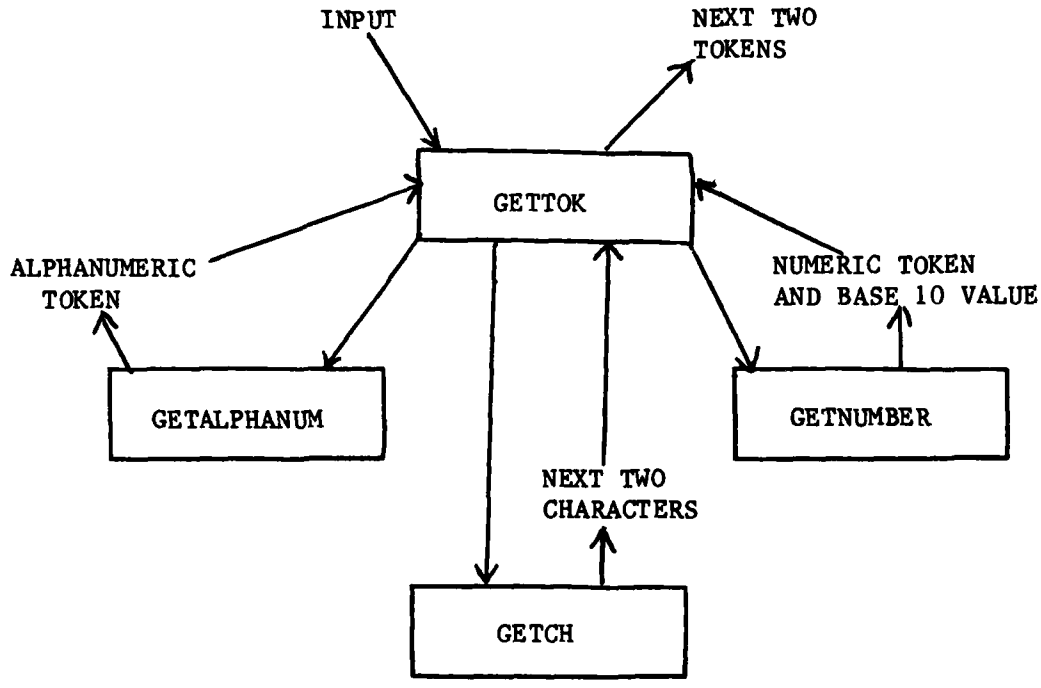


Figure 3-2: Lexical Analysis Structure

GETTOK must return the next two tokens since the parser needs two token lookahead in certain place. It accomplishes this by simple replacing the current token with the next token when it is called and then gets the token following the next token. Essentially **GETTOK** is responsible for identifying the token following the current token.

When the recognizer is initialized, **GETTOK** is called twice to get the first two tokens. After initialization, **GETTOK** is only called by the parser whenever it needs another token.

The lexical analyzer is also responsible for putting the token in a form that can be used to lookup the token in the symbol table.

3.5 Symbol Table

It was determined that the symbol table can play a useful role during syntax analysis. The reserved words can be installed in the symbol table along with their enumeration value during initialization. In this way the lexical analyzer can simple lookup the reserved word and return its token value. A desirable side effect of this is that reserved words are prevented from being used as Ada identifiers, since the parser would immediately know that the current token is a reserved word. This is in accordance with the Ada requirements for identifiers and their use. The symbol table is discussed in more detail in this section.

During syntax analysis the symbol table is used to lookup alphanumeric tokens encountered during lexical analysis. If the given token is a reserved word, the enumeration value of that token is returned. If the token is a predefined object, such as an attribute, certain

information such as its type is conveniently available. To accomplish this, it is necessary to put information into the symbol table during initialization of the recognizer. Symbol Table initialization is perform by the procedures **INITRESWRDS, INITPREDEF** and **INITCHARCONSTS.** As their names imply, these procedures initialize the reserved words, predefined objects, and the character constants, respectively. At this point it is important to discuss the structure of the symbol table.

In its current state the symbol table is a tree type structure. The first level of the symbol table is a hash table. Each entry in the hash table contains a pointer to a chain of nodes. If there are no entries corresponding to a particular hash table entry, the entry contains the value "nil".

The information contained in the nodes pointed to by the hash table are three pointers. The first entry points to the the lexical value for the name of an entry. The second points to a node containing specific information about the object represented by the name. The third entry points to the next node in the chain. If this is the last node in the chain, the pointer is nil. This is best illustrated by the following diagram.

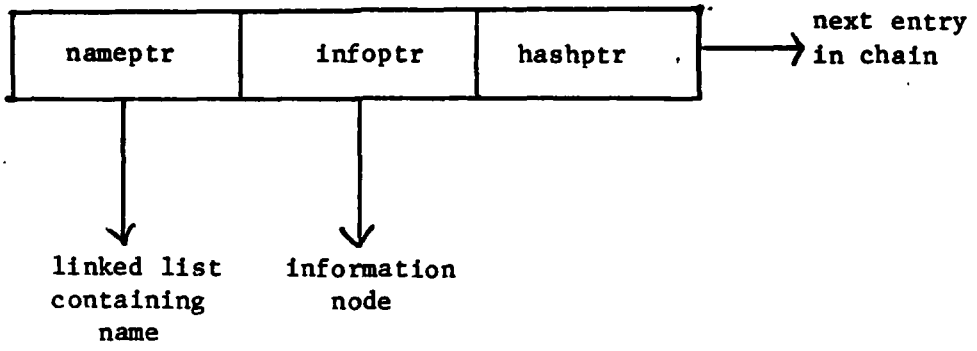


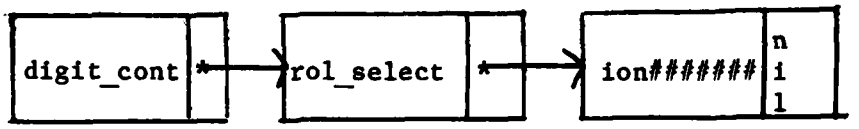
Figure 3-3: Structure of Hash Node

It is appropriate that the name node and information node be discussed in more detail.

3.5.1 Name Node

The name node contains a pointer to the character representation (lexical value) of a name. Entering the character representation of a name into the symbol table is complicated by the requirements of Ada. One of the requirements for Ada identifiers is that all characters in an identifier are significant. This implies that all of the characters of an identifier must be stored. Traditionally, only the first few characters (usually between six and twelve) were significant. To meet this requirement, lexical names in this effort are stored as linked lists. Each node

in the list holds ten characters and a pointer to the next node in the list, if it exists. For example, an identifier such as "digit_control_selection" would be stored as follows.



is the space character

* is the pointer to the next node

Figure 3-4: Name Storage Structure

Note that any empty space after the identifier is specifically padded with blanks. This is to insure uniformity of implementation over various computer systems.

Ada has an additional requirement that upper and corresponding lower case letters are equivalent in identifiers. This was handled by simply transliterating upper case characters to lower case characters, except in the case of character and string literals.

3.5.2 Information Node

The information node contains data that is applicable to the object it represents. Different types of objects require different types of information to be stored in the symbol table about them. For example, the information needed for a reserved word is simply its enumeration value, while the information needed for a predefined object is its type and its size. In order to efficiently implement this scheme, a Pascal variant record is used.

3.5.3 Other Symbol Table Issues

There are two occasions upon which information is entered into the symbol table. These are during initialization and while processing the declarations contained in an Ada program. The latter mainly deal with semantic analysis, therefore no attempt is made to enter information during declaration elaboration. The symbol table is not fully developed since it was assumed that future efforts would require their own symbol table design.

3.6 The Parser

After the initialization and lexical procedures were implemented and tested, the parser was constructed. Construction of the syntax analysis procedures began with COMPILATION_UNIT diagram. COMPILATION_UNIT is the starting symbol for this particular specification of the Ada syntax.

Subsequently, procedures required by COMPILATION_UNIT were constructed, followed by the procedures the other procedures required and so on in top down fashion. For example, COMPILATION_UNIT references the GENERIC_SPECIFICATION diagram. GENERIC_SPECIFICATION references the GENERIC_FORMAL_PARAMETER diagram which in turn requires the SUBTYPE_INDICATION and PROC_FORMAL_PART diagram, among others. These diagrams may be found in the appendix. This process, which was quite tedious and obviously a candidate for automation, was iterated until all of the syntax diagrams were transformed into procedures.

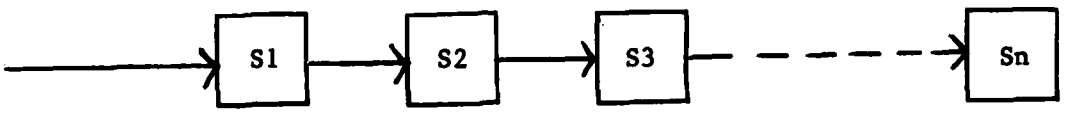
The process of transforming the syntax diagrams into Pascal procedures closely follows that of Wirth [Ref 12]. Basically, it follows a structured set of rules. It is appropriate to define some notation first.

T(S) corresponds to the transformation of the graph S.

L(S) represents the set of initial symbols for graph S.

L(S) = FIRST(S)

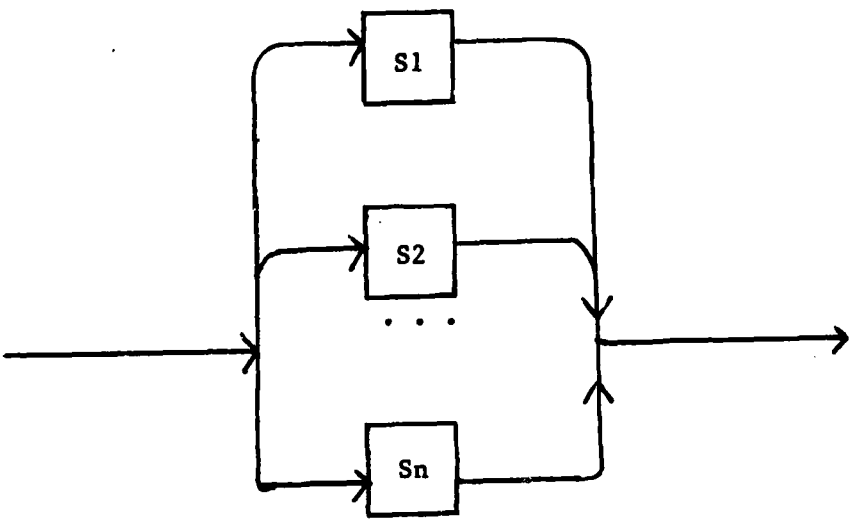
A1. A Sequence of elements:



is transformed to the compound statement:

```
begin
        T(S1);
        T(S2);
        T(S3);
          .
          .
          .
        T(Sn)
end;
```

A2. A choice of elements:

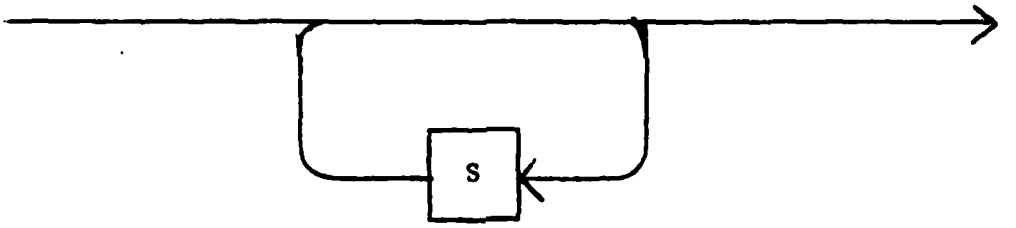


is translated to a case statement.

```
        case token of
L1:     T(S1);
L2:     T(S2);
 . . . . . .
Ln:     T(Sn)
end;
```

where Ln = L(Sn) = FIRST(Sn)

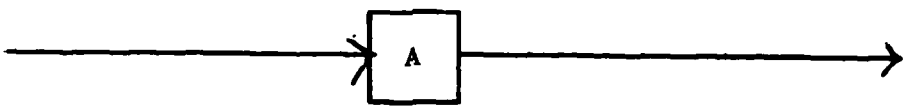
A3. A loop of the form:



is translated to the statement

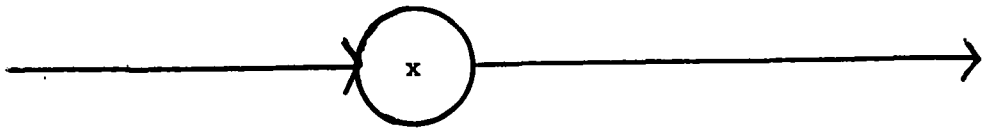
while token in L(S) do T(S)

A4. An element of the graph denoting another graph, A



is transformed into the procedure call for statement A.

A5. An element of the graph denoting a terminal symbol, x



is translated into the statement:

```
if token = x then gettok else error
```

where error is the routine called when an ill-formed construct is encountered.

In addition the frequently occuring construct:

```
gettok;
T(S);
while B do
begin
        gettok;
        T(S)
end;
```

can be expressed as:

```
repeat gettok; T(S) until not B
```

This method was modified slightly, since two token lookahead was required in certain cases.

The next section shall present an example which traces the process of taking a BNF type grammar, constructing the syntax diagrams from it and transforming the diagrams into

Pascal procedures.

3.7 Example

An example of transforming a BNF type grammar into Pascal procedures is presented here. It was previously mentioned that the TYPE DECLARATION diagram was missing from the Bonet set. This shall serve as an appropriate example.

The specification from the Ada Language Reference Manual is the following BNF definition:

```
type_declaration ::=

    type identifier [discriminant_part] is type_definition
  | incomplete_type_declaration


type_definition ::=

    enumeration_type_definition
  | real_type_definition
  | integer_type_definition
  | array_type_definition
  | record_type_definition
  | access_type_definition
  | derived_type_definition
  | private_type_definition


incomplete_type_declaration ::=

    type identifier [discriminant_part] ;
```

Figure 3-5: BNF for TYPE_DECLARATION

The BNF definition for all of the choices for

TYPE_DEFINITION are:

```
enumeration_type_definition ::=
  ( identifier { , identifier } )

integer_type_definition ::=
  range_constraint

real_type_definition ::=
  floating_point_constraint
| fixed_point_constraint

record_type_definition ::=
  record  component_list  end record

access_type_definition ::=
  access  subtype_indication

derived_type_definition ::=
  new  subtype_indication
```

Figure 3-6: BNF Associated with TYPE_DEFINITION

Wirth's method (see section 3.2) can be applied to this formal specification. The resulting diagrams can then be "folded" or collapsed. "Folding" in this context mean to substitute small diagrams that are only used for one specific

construct for their namesake's box in the more global diagram. For example **enumeration_type_definition** can be folded into the **type_definition** diagram by replacing its box in **type_definition** with the diagram for **enumeration_type_definition.** Note that the diagram **constraint** has not been folded into **type_definition** because it is used in other diagrams (see appendix). **type_definition** has not been folded into **type_declaration** because it is a fairly large diagram. Obviously, the choice of folding or not folding is a heuristic one. It depends on the preference of the individual doing the folding. The following diagrams are the result of folding all of the different type definitions into two compact diagrams. These diagrams are shown in figures 3-7 and 3-8.

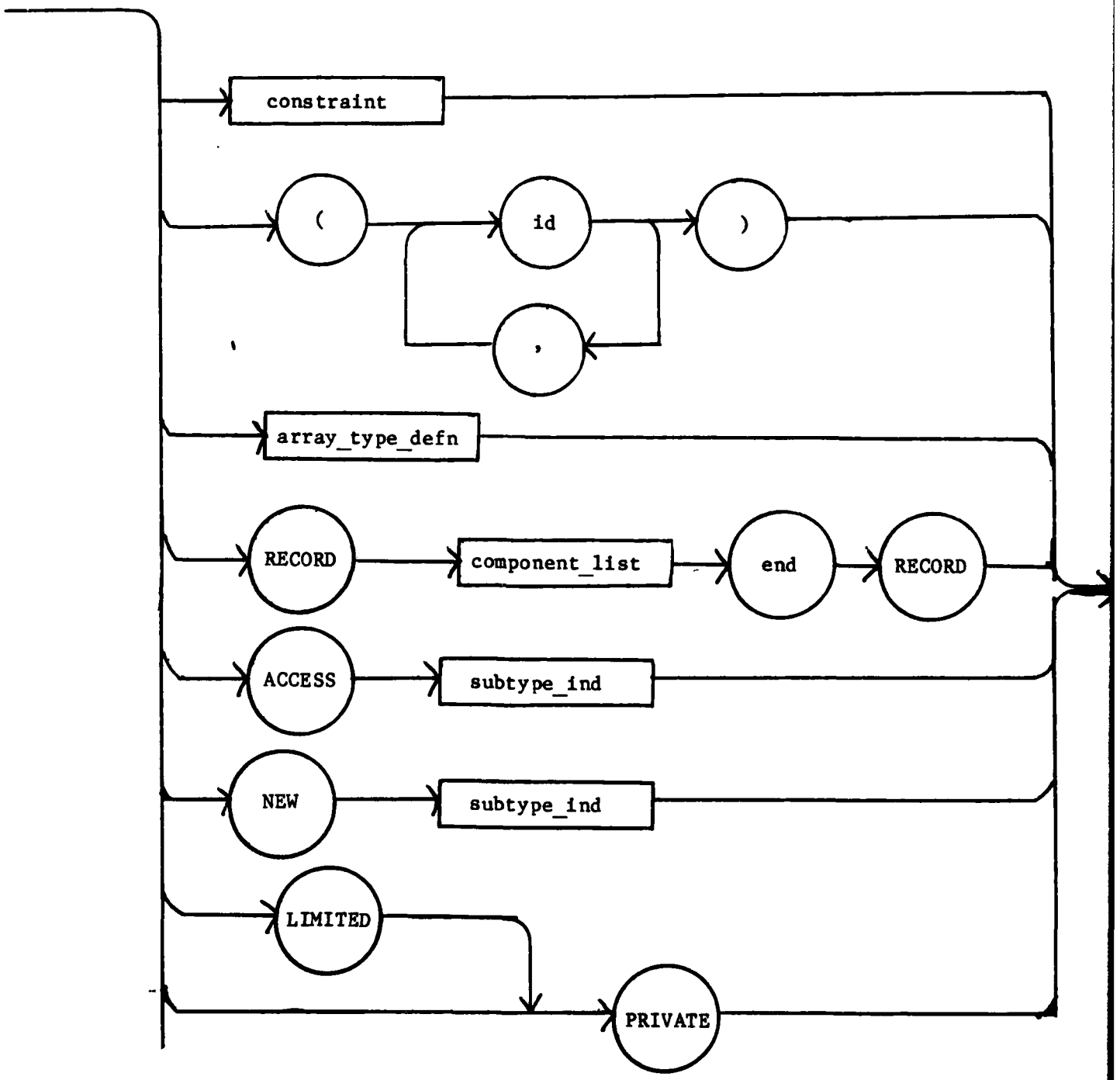


Figure 3-7: TYPE_DEFN Diagram

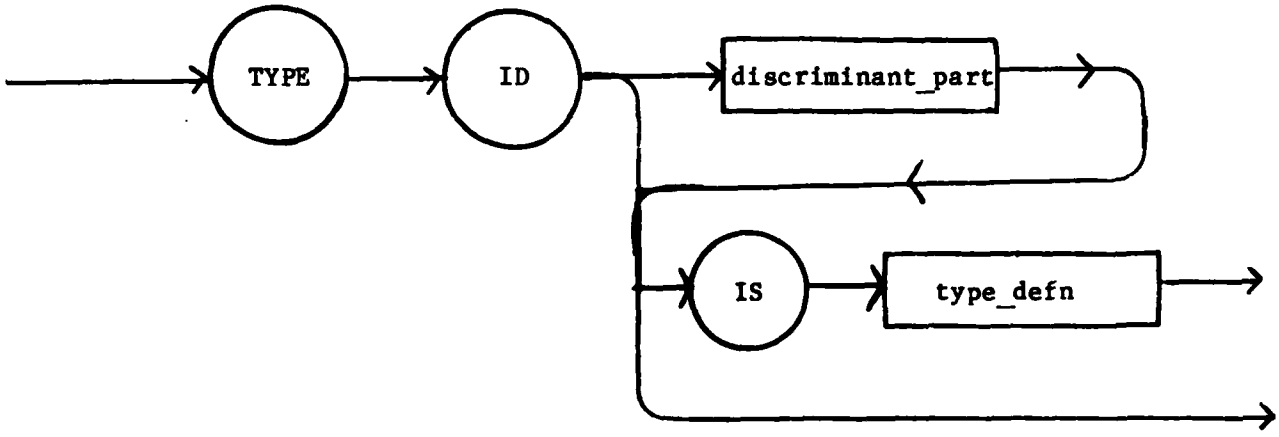


Figure 3-8: TYPE_DECL Diagram

The next step is to transform the diagrams into Pascal procedures. Once again the method used is due to Wirth [Ref 13:508]. The result is the following two procedures:

```
procedure type_decl;
begin
        if token <> typetok then error(0,0,0);
        gettok(token,nexttok);
        getid (typename);
        gettok(token,nexttok);
        if token = lparen then discriminant_part;
        if token = istok then
            begin
                gettok(token,nexttok);
                type_defn;
            end;
end;
```

Figure 3-9: Procedure **type_decl**

```
procedure type_defn;

begin
        case token of

  rangetok,
  deltatok,
  digitstok,
  lparen:        constraint;

  arraytok:      array_type_defn;

  recordtok:     begin
                   gettok(token,nexttok);
                   component_list;
                   if token <> endtok then error (0,0,0);
                   gettok(token,nexttok);
                   if token <> recordtok then error (0,0,0);
                   gettok(token,nexttok);
                 end;


  accesstok:     begin
                   gettok(token,nexttok);
                   subtype_ind;
                 end;

  newtok:        begin
                   gettok(token,nexttok);
                   subtype_ind;
                 end;

  limitedtok,
  privatetok:    begin
                   if token = limitedtok then
                     begin
                         gettok(token,nexttok);
                     end;
                   if token = privatetok then
                     begin
                         gettok(token,nexttok);
                     end
                   else error (0,0,0)
                 end;

   others:       error(0,0,0);
        end; {case}
end;
```

Figure 3-10: Procedure type_defn

3.8 Other Issues

3.8.1 Error Recovery

At this point, no serious attempt at error recovery is made. Simple errors, such as typographical errors and missing semicolons are reported and parsing continues. If there is a more serious error, it is likely that many more error messages will be generated than there are bona fide errors. Therefore when an error is encountered, the parser reports that the program is invalid and terminates. It is acknowledged that this is not an elegant way to deal with this situation, but the goal of this effort is to accept syntactically correct Ada programs and reject those that are invalid.

Wirth [Ref 13:508] addresses this issue. A future effort might involve incorporating Wirth's scheme into this effort.

3.8.2 Software Engineering Concerns

The software for this thesis was developed with an attention to software engineering principles. The overall design of the recognizer is modularized. There are no **GOTO** statements in the software at all. The names chosen for procedure names, identifiers and other object convey their purpose. Therefore, documentation of every line of code is not necessary. There should be no problem following the

program if the syntax diagrams are available.

4. Evaluation

The chief use of the Ada recognizer would likely be as a programmer's tool. This tool can have many uses. It can be used as a basis for a syntax directed editor, the basis for a compiler or as a syntax verifier.

4.1 Syntax Directed Editor

A syntax directed editor does not allow the creation of a malformed program. This type of editor knows in advance what the possible choice of tokens are in a particular situation. For example, an Ada program must start with one of the following reserved words: **PRAGMA**, **WITH**, **PROCEDURE**, **FUNCTION**, **PACKAGE**, **GENERIC** or **SEPARATE**. The editor would allow the programmer to select one of these reserved words and no others at that point. Once the programmer selected the desired keyword, the editor knows how the selected construct should be formed. For instance, **PROCEDURE** and **FUNCTION** must be followed by an identifier.

The editor would also be able to build a symbol table for the program. It would be able to consult the symbol table to determine if a particular function or identifier has been declared. If it has, semantic information about this entity can be used to determine if a statement is semantically correct.

The syntax directed editor is a programming tool used in a programming support environment. Programs written using a syntax directed editor are guaranteed to be syntactically correct. Certain semantic errors can also be detected. Creating a program using this type of editor accomplishes some of the work a standard compiler would need to do. The only activities remaining would be the completion of semantic analysis and code generation.

In order to effectively implement a syntax directed editor from the Ada recognizer, several modifications are needed. One of these modifications is to make the input more interactive. The user would have to be told what his possible options are for the next input symbol. Presently, the input is just taken from a file. This is analogous to batch mode. In addition, the input routines would be required to be able to erase erroneous constructs as well as individual characters. The input routines would also be required to let the user leave constructs incomplete if the need arose.

In general, the input routines would need to make the use of the editor as pleasant as possible for the user.

4.2 Syntax Verification

The Ada recognizer can also be used as a filter for syntax errors. In most cases code generation is a waste of

valuable computer time if the source program contains syntax errors. If the source program is run through the Ada recognizer first, the syntax errors can be located and corrected before compilation.

Modifications to the recognizer can be made to suspend syntax verification and allow the user to correct an error. Once the error is corrected, syntax analysis can continue. The corrected program can then be written to disk and the erroneous program disposed of.

In this case, the modifications are similar to those need for the syntax directed editor. These are the ability to ignore erroneous symbols from the input stream and substitute symbols from the users terminal.

This technique is justified by the fact that any full Ada compiler would consume a large amount of computer time, since the language is large and the code generation is complex.

Semantic analysis and code generation are not incorporated into this effort. The next chapter discusses how they could possibly be implemented.

5. Recognizer to Compiler Transformation

The Ada recognizer can be transformed into an Ada compiler by the addition of procedures to handle symbol table management, semantic analysis and code generation.

5.1 Symbol Table

The remainder of the symbol table must be designed. This would include creating templates for the nodes to store information about identifiers, arrays, procedure, functions, types, subtypes, parameters, packages, generic units, etc.

More specifically, the information for an identifier would include the type, scope and constraints.

The information needed for procedures and functions are nearly identical. The information required is the number of parameters, the parameters themselves, scope information and in the case of functions, the type.

The information needed for arrays is the type, dimension, indices, the type of the indices, and the bounds of the indices

The information required for types are its basic type (array, integer, real, character, etc.) and any constraints.

5.1.1 Symbol Table Management

Information is entered into the symbol table only during initialization or while processing program declarations. The initialization phase is already implemented. While the declarations are being processed it will be necessary to build the information node to install in the symbol table.

```
XYZ : INTEGER
```

Figure 5-1: Example Declaration

The declaration in figure 5-1 is parsed by the routine **obj_num_exc_decl** which is partially reproduced here.

```
procedure obj_num_exc_decl;
begin
        if token <> id then error();    {1}
        getid(idptr)                    {2}
        gettok(token,nexttok);          {3}
        case token of                   {4}

colon:  begin                           {5}
        gettok (token, nexttok);        {6}
        case token of                   {7}

        arraytok:       . . . .         {8}

          .

          .

          .

        others:         begin           {9}
                        getname(type_mark);  {10}

        .

        .

        .
```

Figure 5-2: Parser Code for Declaration

The statement labeled {1} checks to see if the current token is an identifier. If it is not, an error has occured. Statement {2} returns a pointer to the node for this identifier in the symbol table. Statement {3} gets the next token, in this specific case it should be a colon. Statement {4} selects the appropriate path for the parser to take based on the current token. In this case the only two

possibilities are colon and comma. The path for comma is not shown in the example. Statement {6} reads past the colon and returns the next token. At this point there are four legal possible tokens. These are **array**, **exception**, **constant** and a type name. Statement {7} chooses the path based on the current token. Statement {10} gets the name of the specific type. The case statement uses the others option, since the type name could be any name.

The code necessary to enter the appropriate information into the symbol table can be incorporated directly into the procedure that recognizes this construct. The resulting procedure would look like this:

```
begin
        if token <> id then error();
        getid(idptr);
        new (symptr);                   <<--- {1}
        gettok(token,nexttok);
        case token of

colon:  begin
        gettok (token,nexttok);
        case token of

        arraytok: . . . . .

          .

          .

          .

        others: begin
        getname(type_mark);
        symptr^.class:= ident;          <<--- {2}
        symptr^.idtype:= type_mark      <<--- {3}
        install(symptr);                <<--- {4}

        .

        .

        .

end;
```

Figure 5-3: Symbol Table Code Added

The additions are highlighted by the arrows (<<---). The new statement labeled {1} gets a new information node. Statement{2} tells the variant record that the information will be for an identifier. Statement {3} enters the type of the identifier into the information node. Statement {4} puts

the information node into the symbol table.

5.2 Semantic Analysis

Incorporating semantic analysis into the Ada recognizer will probably require the most effort. The specification for Ada semantics is contained in the Language Reference Manual [Ref 11]. Perhaps a better source in terms of implementation is the Ada Compiler Validation Implementers Guide [Ref 6].

Semantic analysis should insure that the program is meaningful. A large part of semantic analysis involves type checking. For example, in Ada, all of the variables and constants in expressions must be of the same type. Explicit type conversions is required when the use of differing types are used. An example of this is adding real numbers and integers together.

Type checking is required in many other instances. It must be verified that the parameters used in a procedure of function call are the same as the formal parameters in the declaration. Array references also require type checking. The indexes for an array must be a discrete type and must match the type of the array declaration.

In some case, semantic analysis code would simply look up the type of an object in the symbol table and compare it to the context of the object at and. Other times it may be

necessary to build a semantic stack to determine the type of a semantically complex expression or statement. An example of this would be a boolean expression used in an IF statement (IF (((A and B) or (not C)) xor D) THEN <statement>).

As with the symbol table management code, the semantic routines can be incorporated directly into the Ada recognizer.

5.3 Code Generation

For the purposes of discussion, it is assumed that the target machine will be Garlington's Ada pseudo-machine. This is appropriate, since the original intent of this thesis was to build a compiler for that machine. The Ada pseudo-machine is a stack machine very similar to Wirth's PL/0 processor.

Code generation can consist of one procedure with three parameters. This procedure would be responsible for emitting assembler instructions for the Ada-pseudo machine. There are three field in an Ada machine instruction, therefore three parameters are needed. These parameters are the instruction, level and address. The key, of course would be to place the calls to this procedure in the appropriate places in the Ada recognizer. A majority of the time this procedure will be called after semantic analysis of a portion of the program. This method has been used by Garlington and Werner.

A simulator for this machine exists. It is written in Pascal and is incorporated into the AFIT-Ada compiler. It can very easily be incorporated into this effort.

6. Conclusions

6.1 Current Status

In its current state the Ada recognizer is relatively complete. Unfortunately, due to time constraints, it remains largely untested. The objective of this testing should be to determine whether or not the Ada recognizer performs its intended function. Test cases, therefore, should include both legal and illegal Ada programs. A good source of test cases is the Ada Compiler Validation Capability suite (ACVC). There are several different classes of errors tested. There are a large number of test programs that contain syntax errors. These should be used to verify that the recognizer rejects invalid programs. There is another class of test programs that only contain errors that can be detected during semantic analysis, at link time or during run time. These can be used to determine if the Ada recognizer accept syntactically valid programs.

The procedure **GETNAME** remains unimplemented. This is because the choice of paths in this routine depends in large part on semantic information. This is also the case with **GETID**. The intended purpose of this routine was to determine the type of a particular identifier.

Semantic analysis and code generation are not

incorporated into this effort. The next chapter discusses how they could possibly be implemented.

6.2 Recommendations

Any follow on effort to the Ada recognizer should address the following issues. First, the recognizer should be thoroughly tested. As previously mentioned, this can be done using the ACVC test suite. Any program errors that are detected at this point should be corrected.

Secondly, the design of the symbol table should be completed. This will depend largely on how semantic analysis and code generation will be performed, so it is suggested that these effort should be performed in parallel.

If the recognizer is to be incorporated into the AFIT-Ada compiler, the structure the of the AFIT-Ada compiler should be examined carefully. Most of the code necessary to manipulate the symbol table and perform semantic analysis is present in the AFIT-Ada compiler. In order to incorporate this work into that compiler, the table driven parser has to be replaced with the Ada recognizer. There is one large case statement in the AFIT-Ada compiler called SEMANTIC. The choices for the case statement correspond to productions in the Ada reference grammar. It simply becomes a matter of finding the proper place in the Ada recognizer to put the code for a particular production.

6.3 Conclusion

This effort represents a necessary first step towards the implementation of a recursive descent Ada compiler. Semantic analysis and code generation procedures must be added.

This is an alternative to the table driven compiler. It is much easier to understand since the syntax analysis procedures are explicitly defined. This allows a reader to determine what the parser is doing and what state it is in with much less difficulty. The addition of semantic procedure and code generation procedures is easier as well. A future implementer would simple place these routines in their proper place in the parser.

As previously mentioned, the parser remains largely untested. However, software engineering practices and a structured method were used to construct the parser, so it can be assumed that the parser is reasonably sound.

The actual coding of the parser took a great deal of time. This is a very tedious process and a prime candidate for automation.

This effort can be used in a variety of ways as mentioned in chapter 4. There are probably more uses for the parser, but it will remain up to some creative individual to

discover them.

BIBLIOGRAPHY

1. Aho, Alfred V. and Jeffrey D. Ullman. Principles of Compiler Design. Addison-Wesley Publishing Company, Reading, Massachusetts, 1979.

2. Barrett, William A. and John D. Couch. Compiler Construction: Theory and Practice. Science Research Associates, 1979.

3. Bonet, R. et al. Ada Syntax Diagrams for Top-Down Analysis. SIGPLAN NOTICES 16, 9 (September 1981), 29-41.

4. DeRemer, F., T. Pennello, and W. M. McKeeman. Ada Syntax Chart. SIGPLAN NOTICES 16, 9 (September 1981), 48-59.

5. Garlington, Alan R. Preliminary Design and Implementation of an Ada Pseudo-Machine. Master Th., Air Force Institute of Technology, March 1981.

6. Goodenough, John B. Ada Compiler Validation Implementers' Guide. TR 1067-2.3 AD-A091760, SofTech, Inc., October, 1980. DARPA/IPTO

7. Goodenough, John B. The Ada Compiler Validation Capability. Computer 14, 6 (June 1981), 57-64.

8. Goos, G. and Wm. A Wulf, editors. Diana Reference Manual. Carnegie-Mellon University and Univeritaet Karlsruhe, March, 1981.

9. Ichbiah, J. D. et al. Rationale for the Design of the ADA Programming Language. SIGPLAN NOTICES 14, 6 (June 1979), 1-1,15-12.

10. Milne, R. and C. Strachey. A Theory of Programming Language Semantics. Chapman and Hall, Ltd., London, 1976.

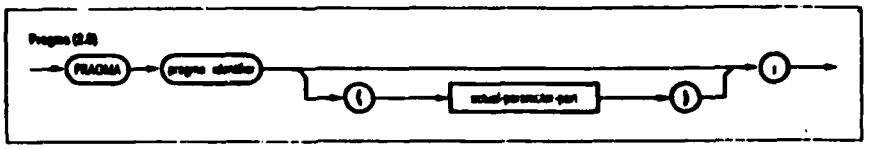
11. , United States Department of Defense. Reference Manual for the Ada Programming Language, July, 1980.

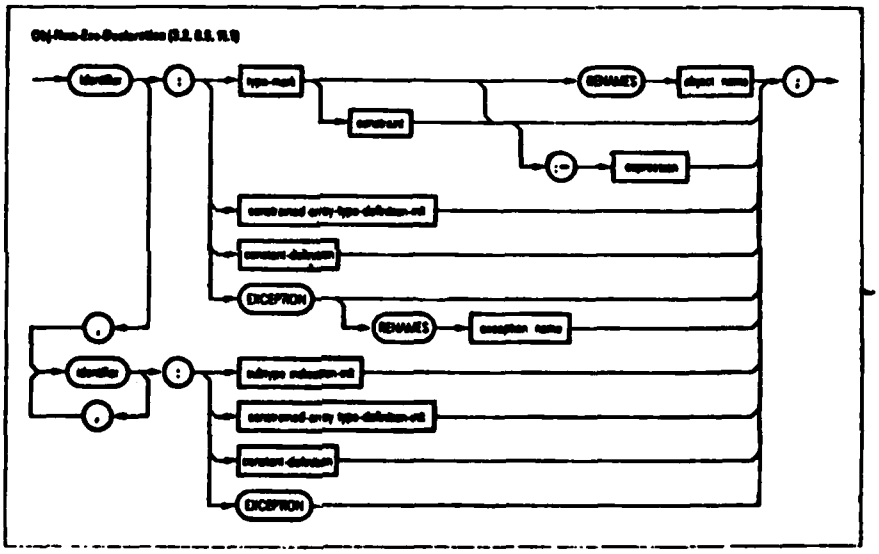
12. Werner, Patrick R. Toward Ada: The Continuing Development of an Ada Compiler. Master Th., Air Force Institute of Technology, December 1981.

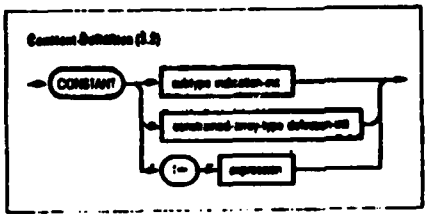
13. Wirth, Nicklaus. Algorithms + Data Structures = Programs. Prentice-Hall, Englewood Cliffs, N.J., 1976.

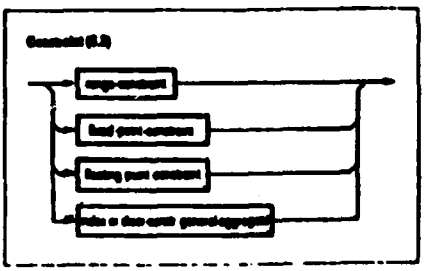
A. Bonet Syntax Diagrams

The Bonet syntax diagrams are included here for reference.

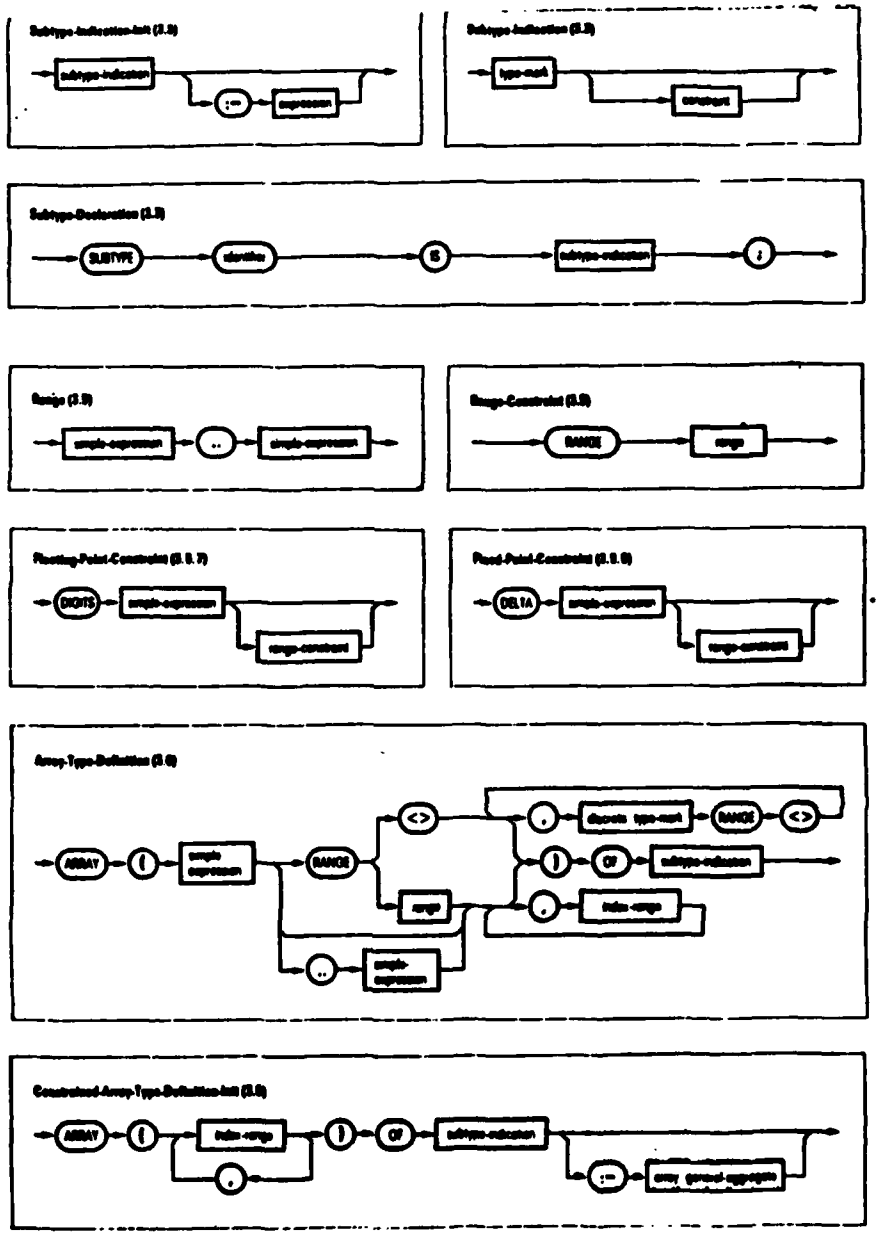




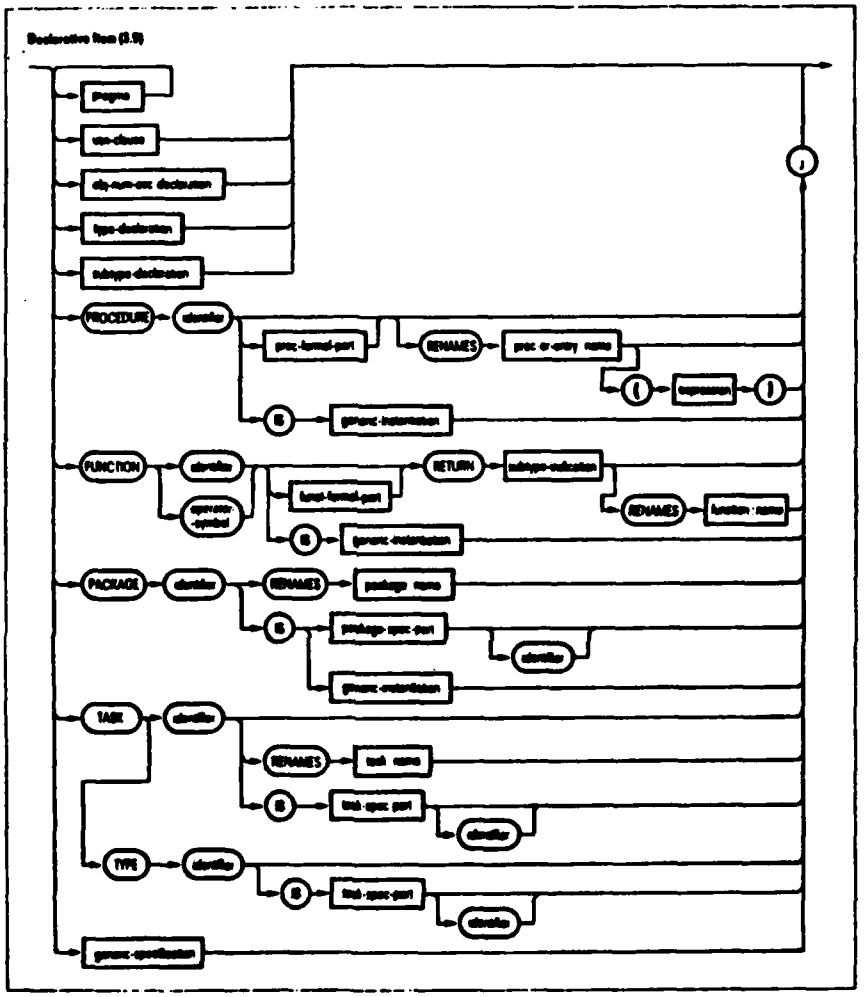




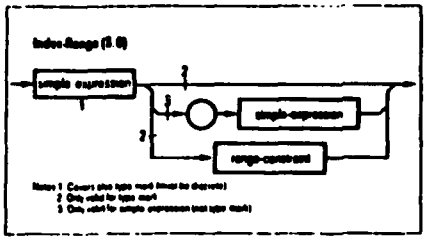
Copy available to DTIC does not permit fully legible reproduction

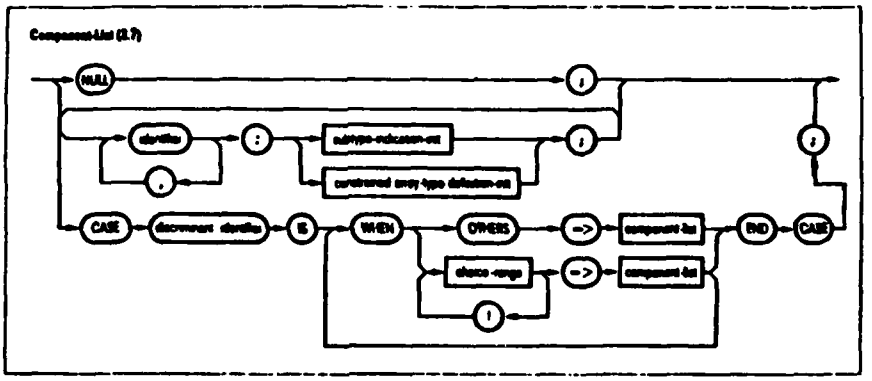


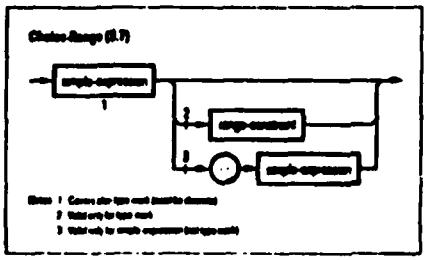
Copy available to DTIC does not permit fully legible reproduction

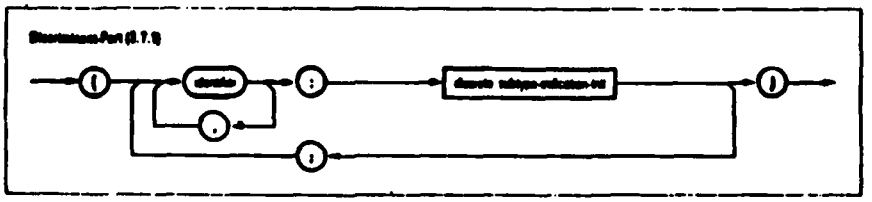


Copy available to DTIC does not permit fully legible reproduction

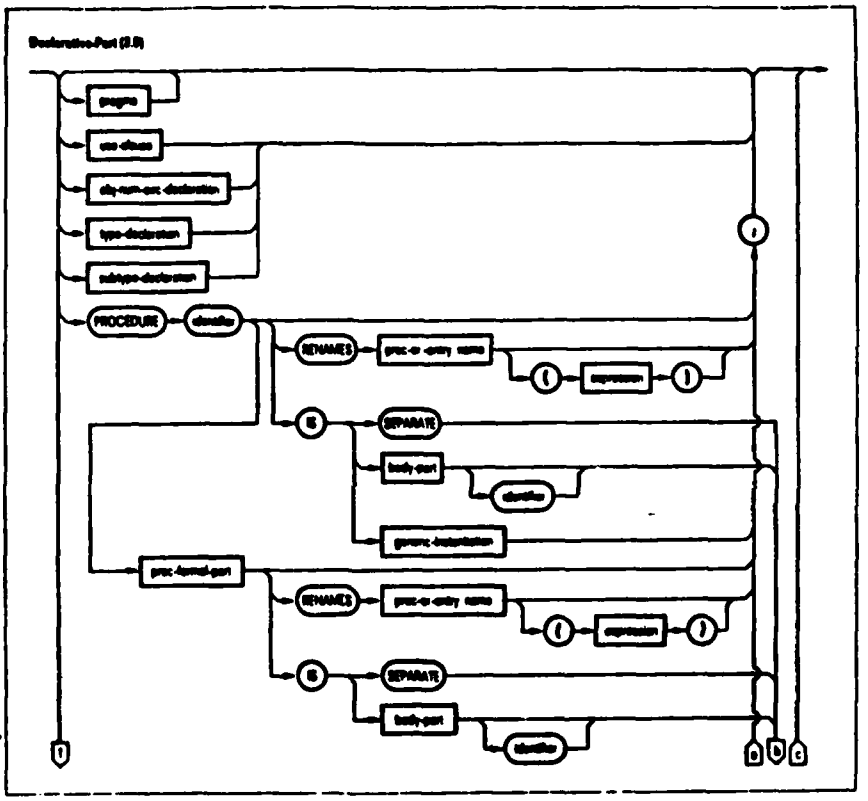




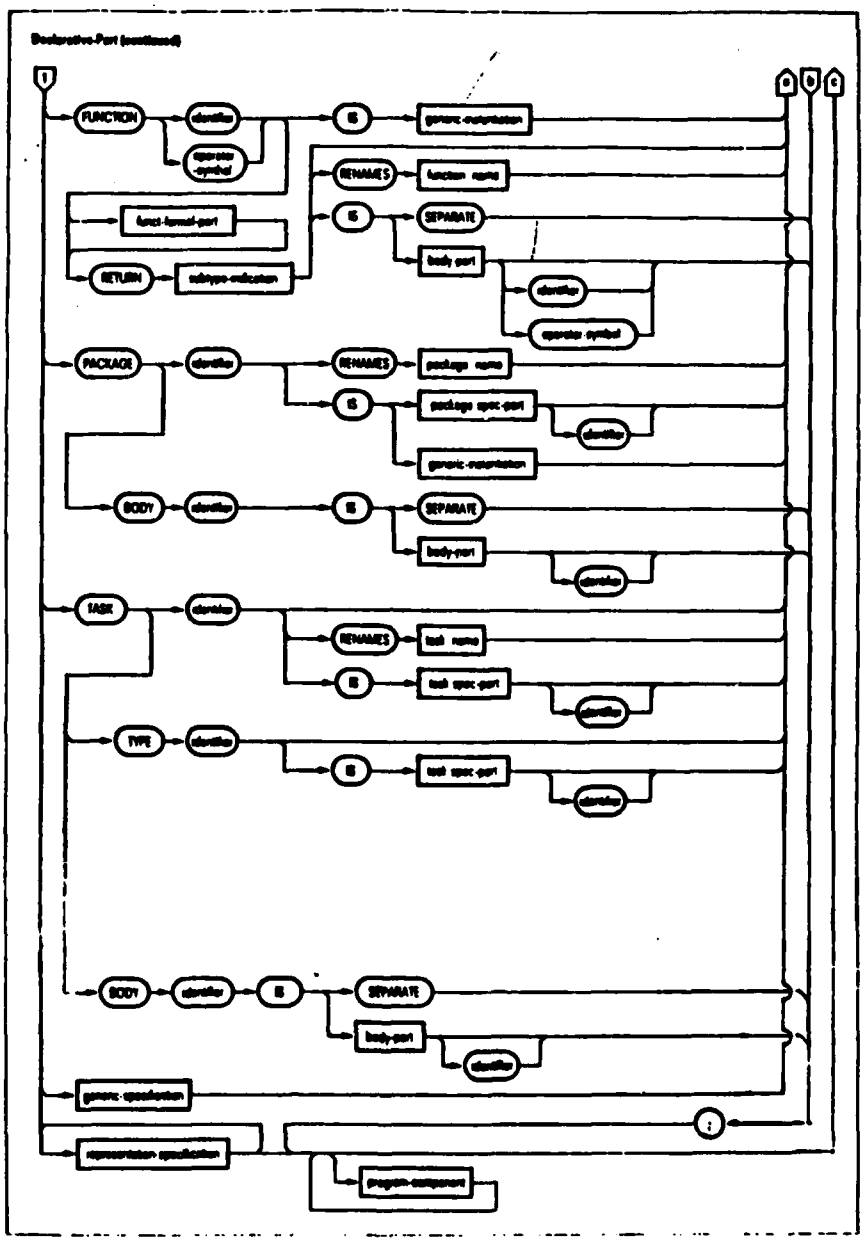




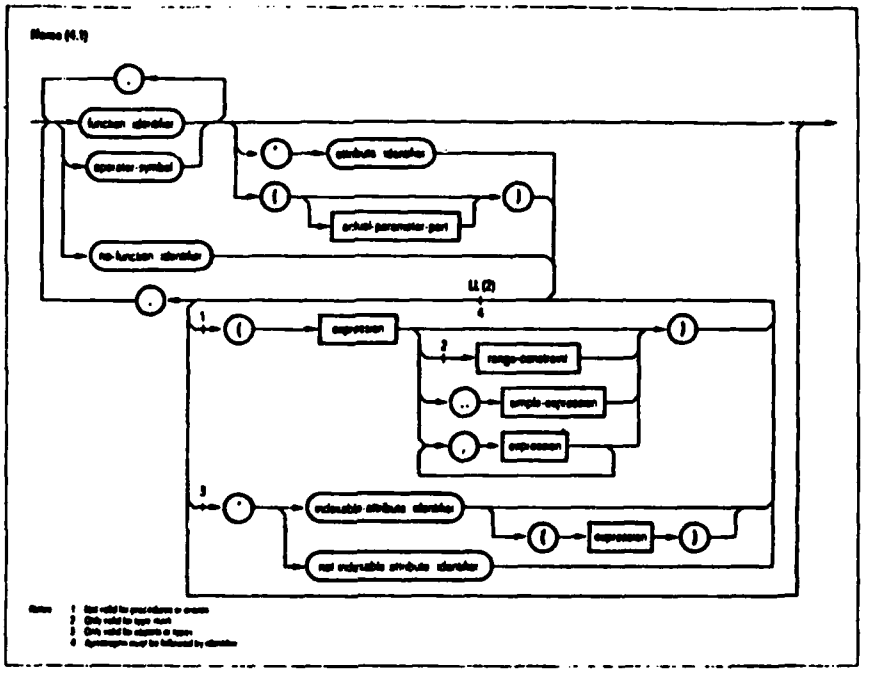
Copy available to DTIC does not permit fully legible reproduction



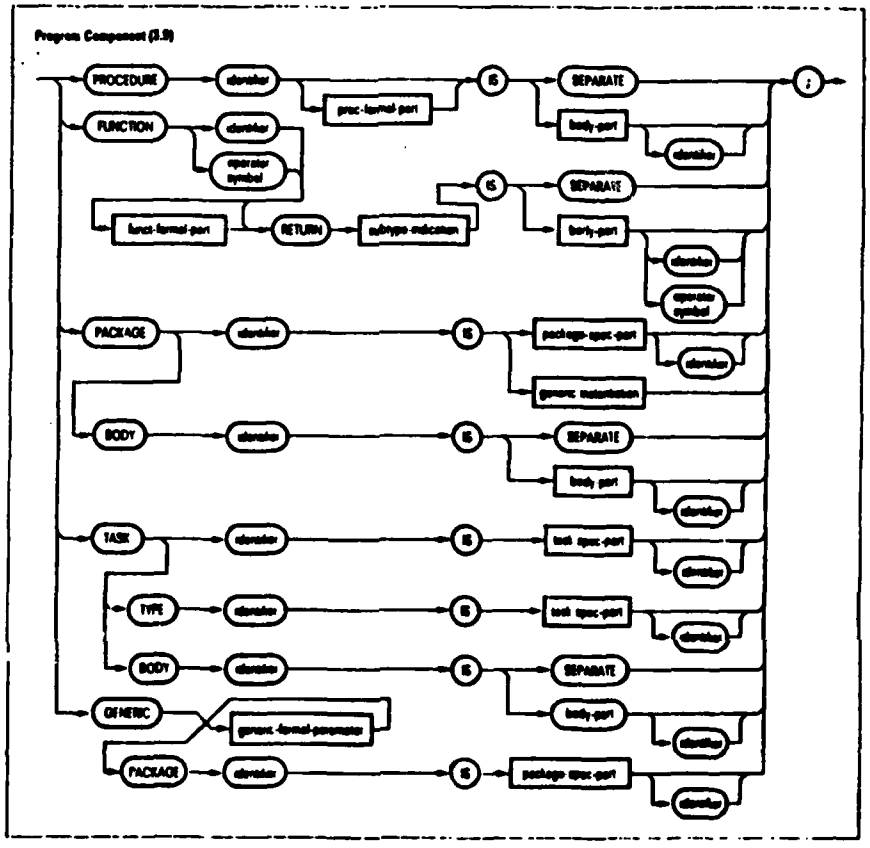
Copy available to DTIC does not permit fully legible reproduction



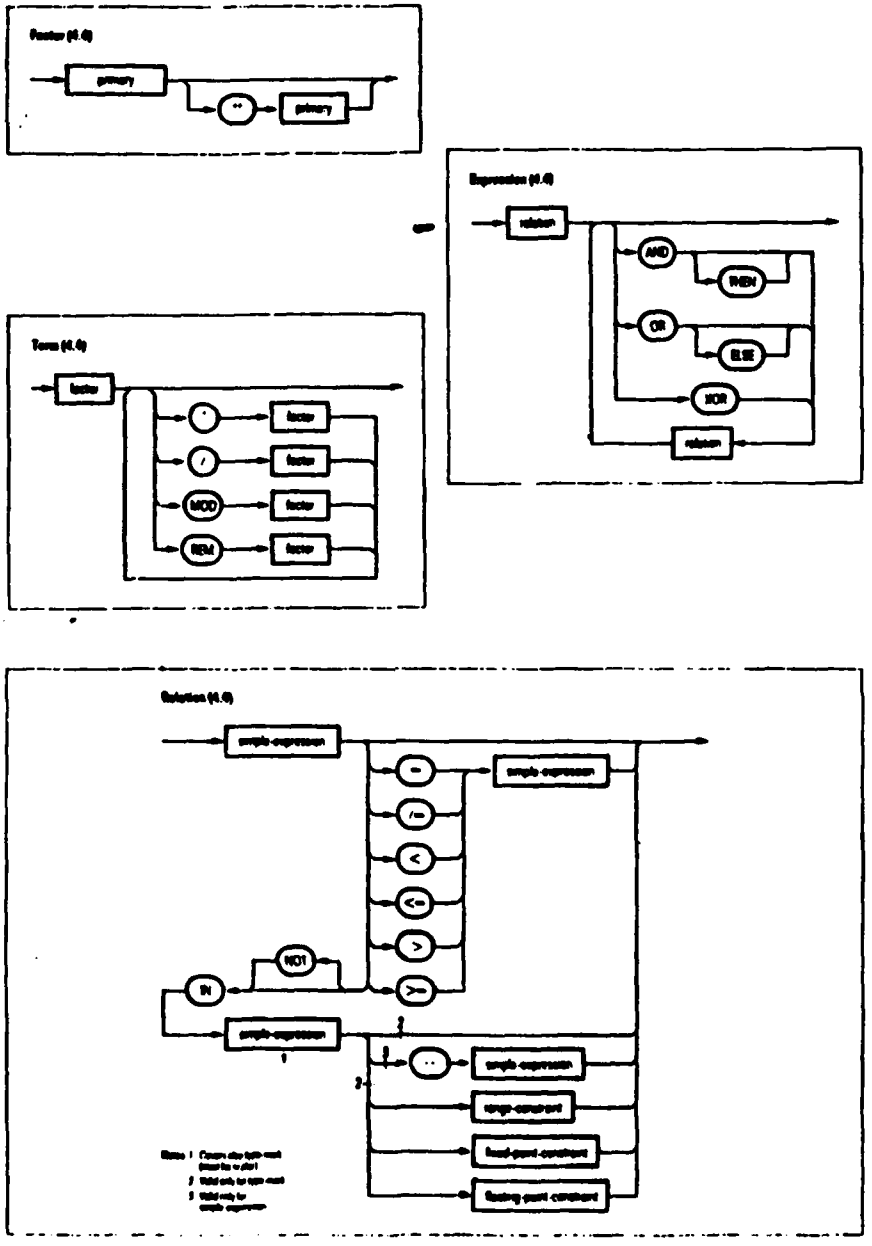
Copy available to DTIC does not permit fully legible reproduction



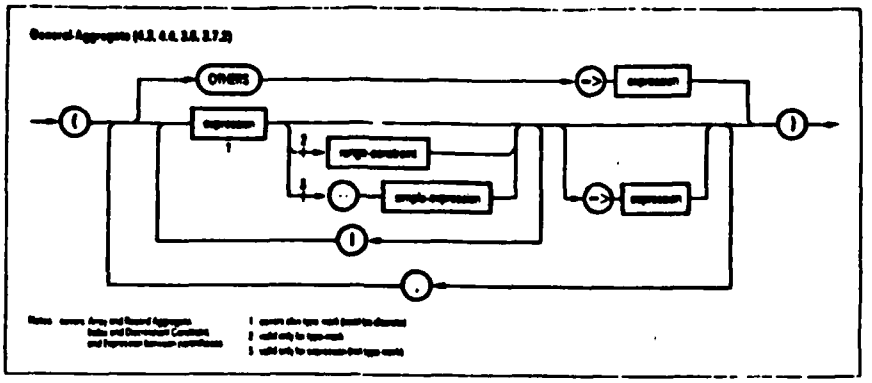
Copy available to DTIC does not permit fully legible reproduction

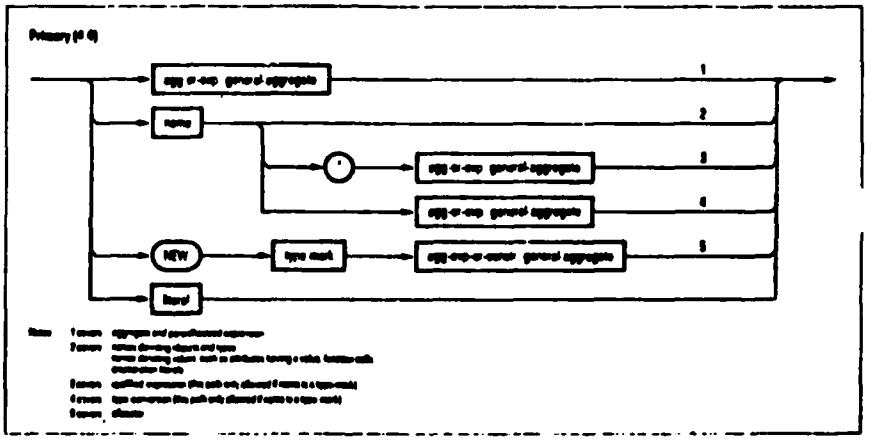


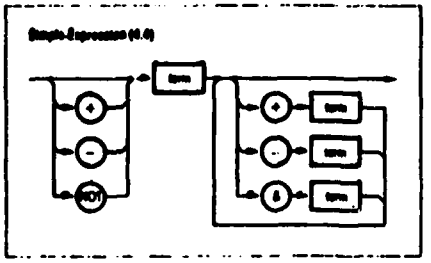
Copy available to DTIC does not permit fully legible reproduction



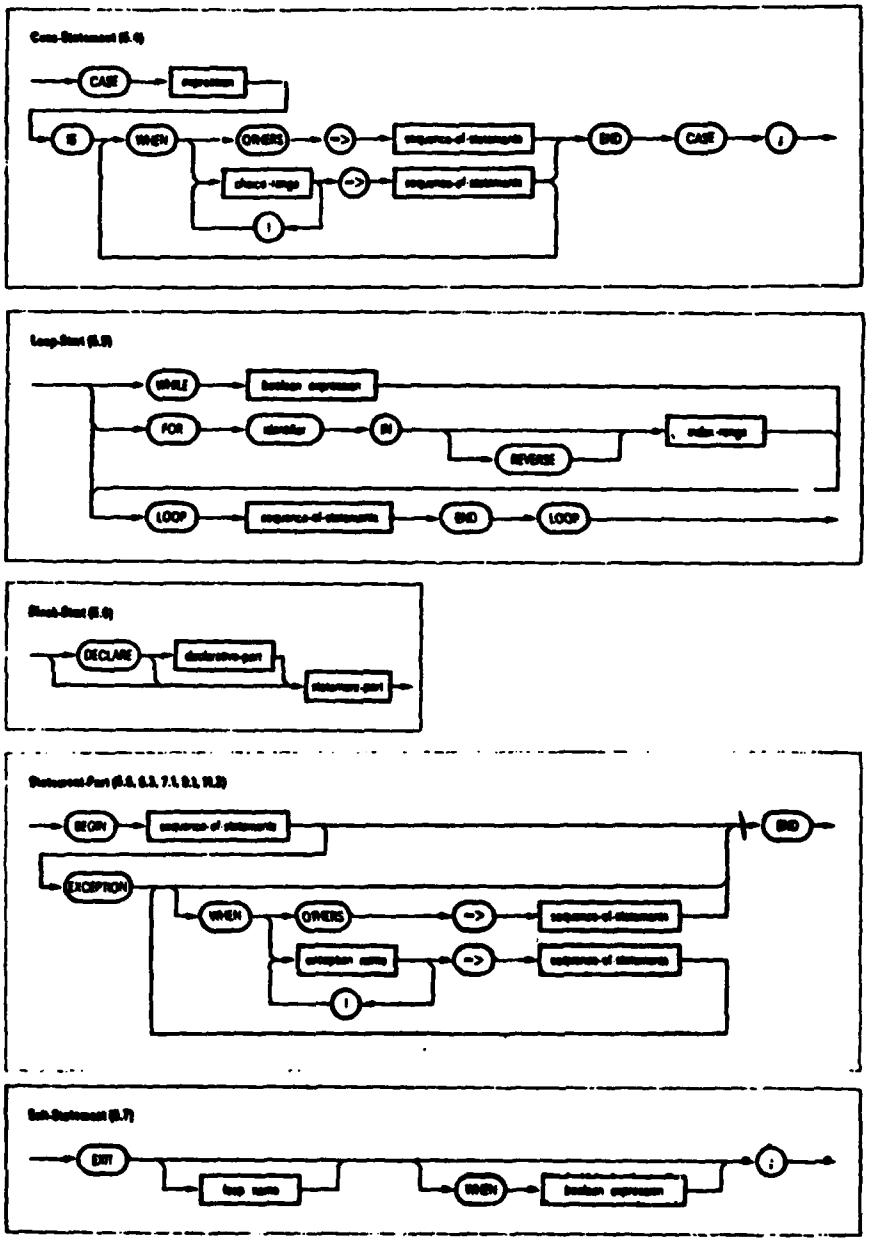
Copy available to DTIC does not permit fully legible reproduction

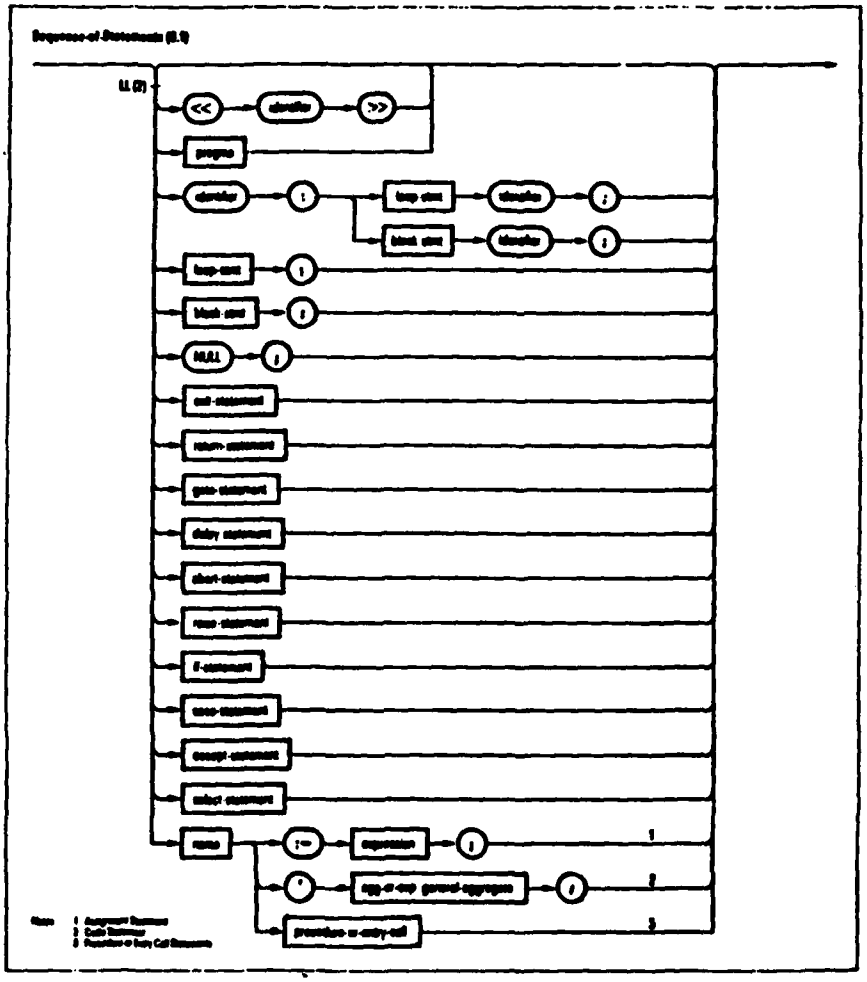


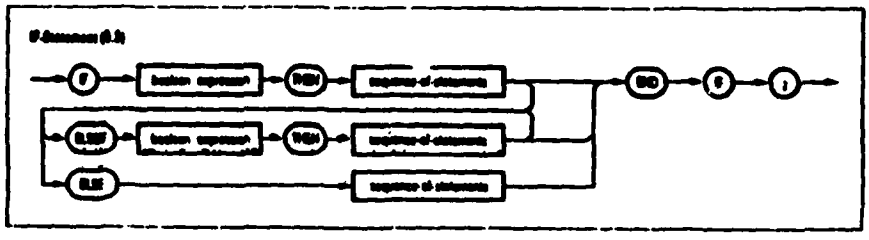




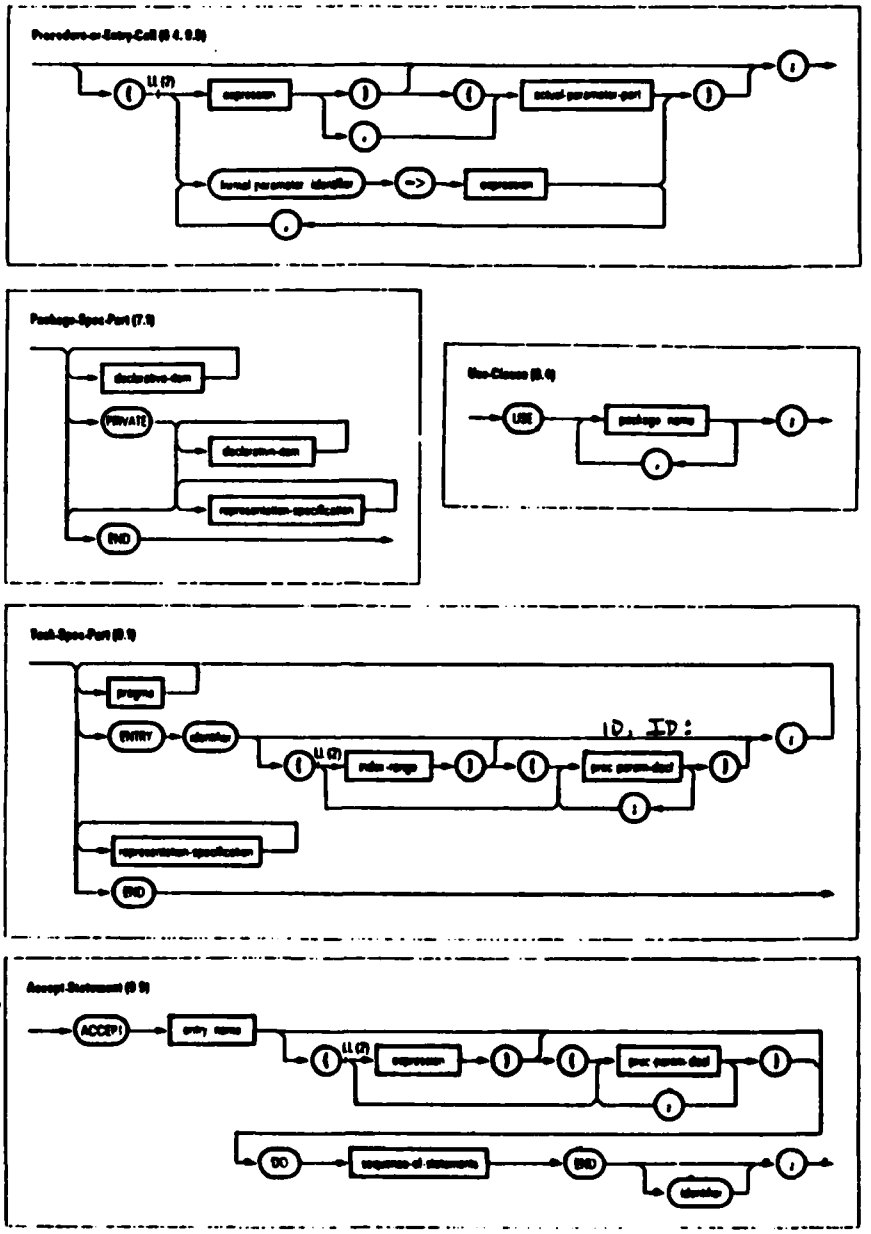
Copy available to DTIC does not permit fully legible reproduction


CASE
IS
WHEN
OTHERS
->
END
CASE
;
|
WHILE
FOR
IN
REVERSE
LOOP
END
LOOP
DECLARE
BEGIN
EXCEPTION
WHEN
OTHERS
->
EXIT
WHEN
;

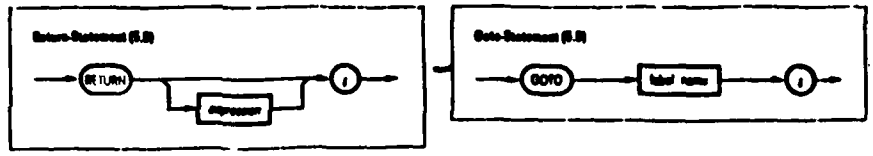


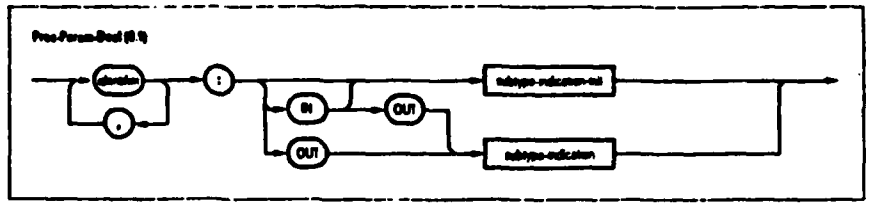


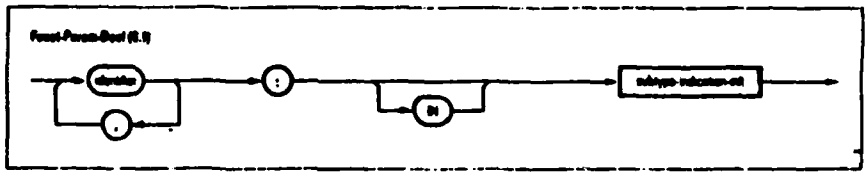
Copy available to DTIC does not permit fully legible reproduction

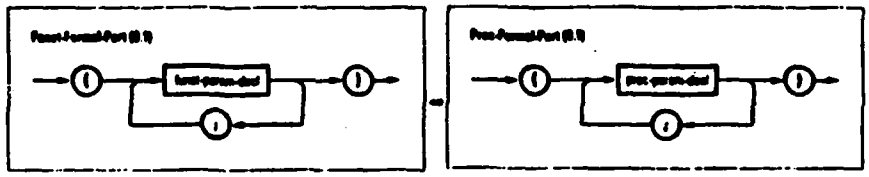


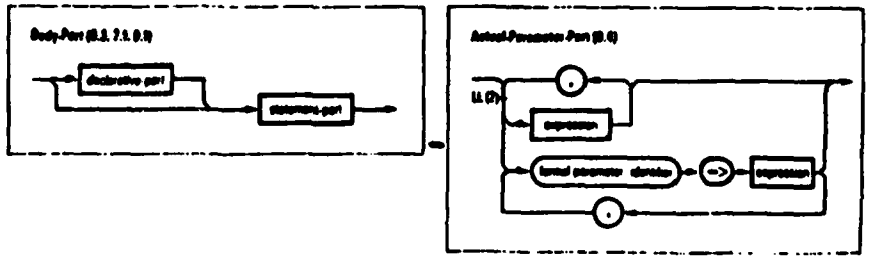
Copy available to DTIC does not permit fully legible reproduction



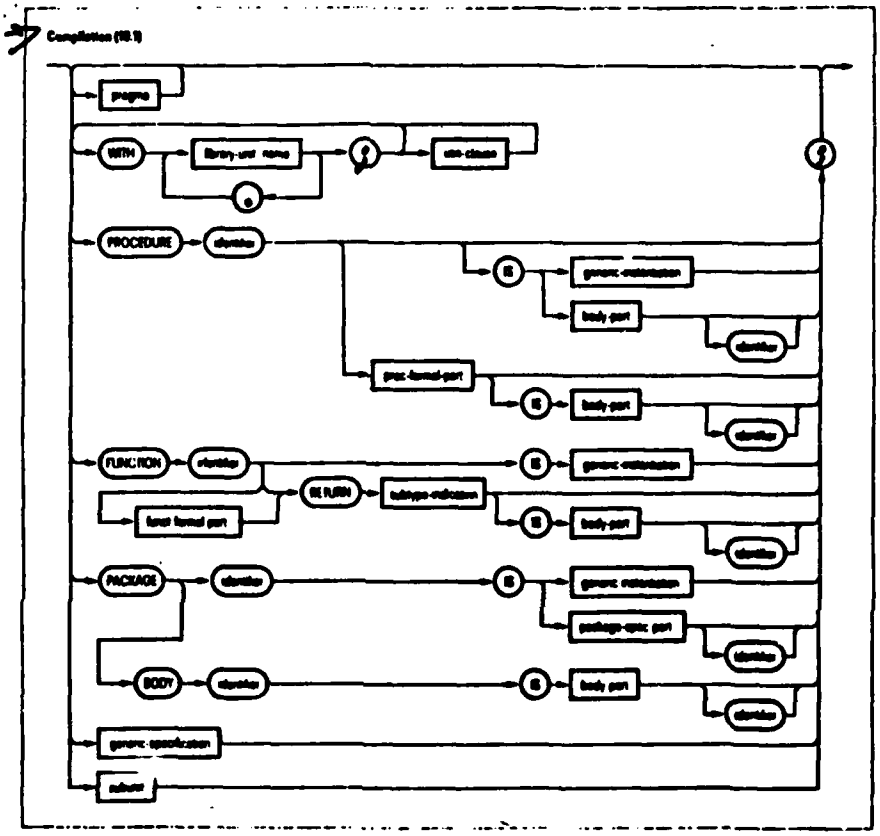




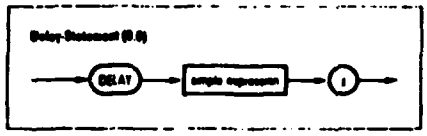


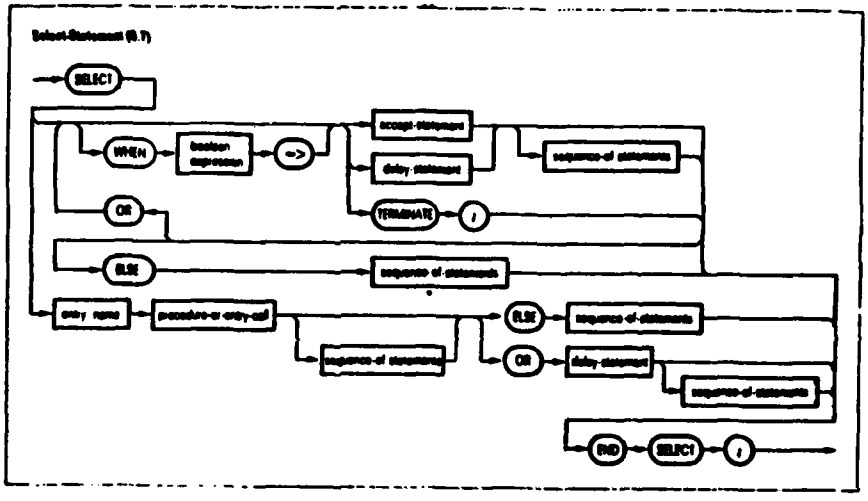


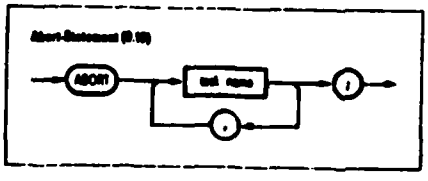
Copy available to DTIC does not permit fully legible reproduction



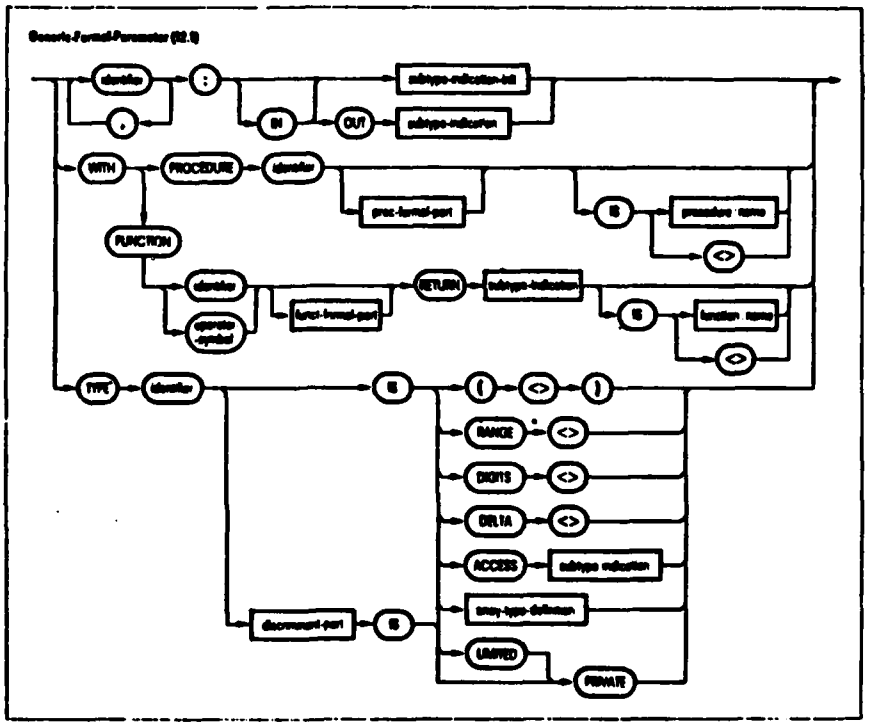
Copy available to DTIC does not permit fully legible reproduction

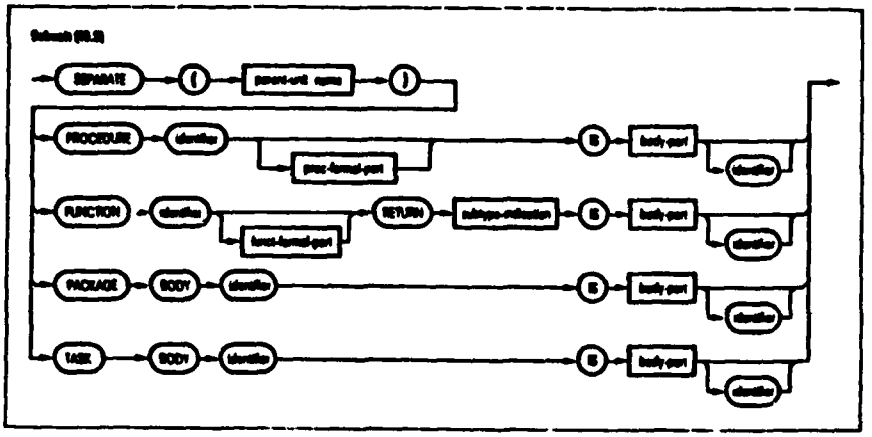


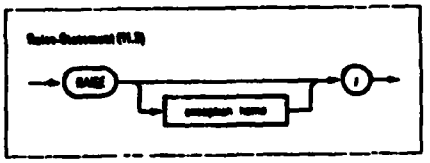


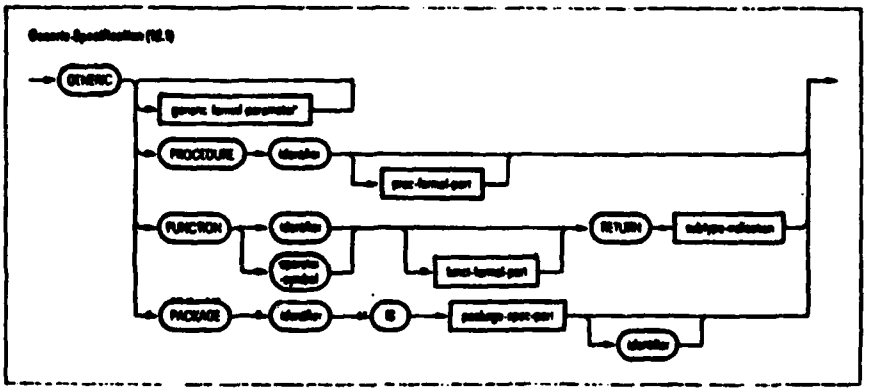


Copy available to DTIC does not permit fully legible reproduction

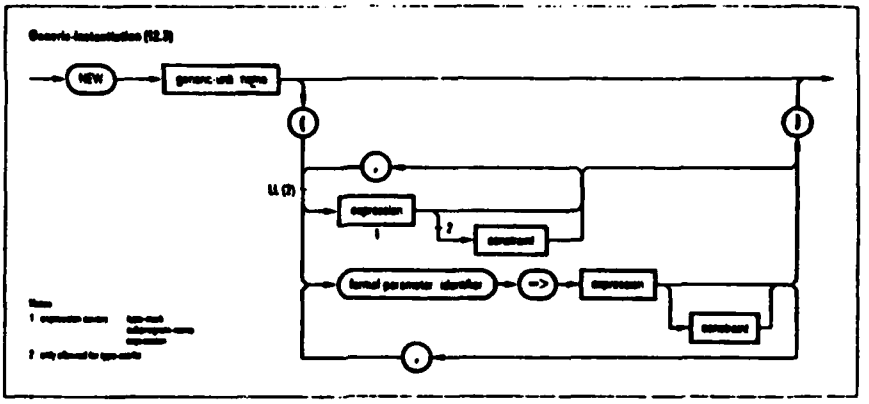

identifier
:
IN
OUT
WITH
PROCEDURE
identifier
IS
<>
FUNCTION
identifier
RETURN
IS
<>
TYPE
identifier
IS
(
<>
)
RANGE
<>
DIGITS
<>
DELTA
<>
ACCESS
LIMITED
PRIVATE

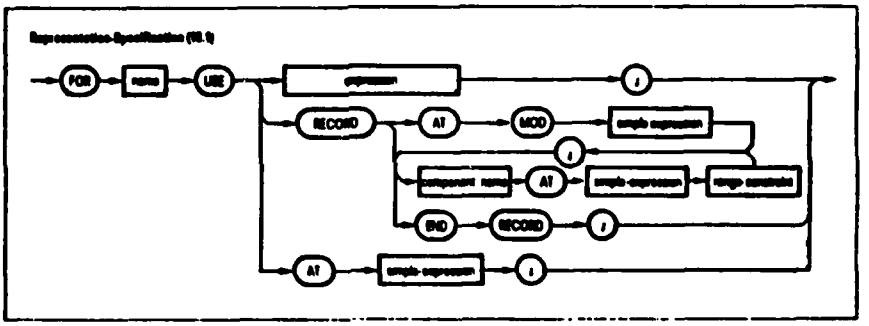






Copy available to DTIC does not permit fully legible reproduction





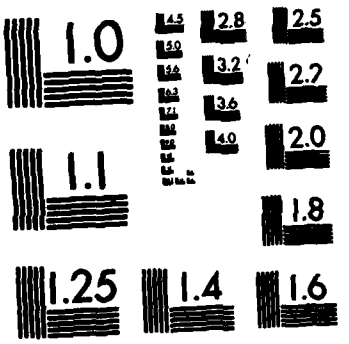
Copy available to DTIC does not permit fully legible reproduction

VITA

William R. Ure was born on 2 May 1959 in Bethpage, L. I., New York to Albert E. Ure and Joyce R. Ure. In 1977 he graduated from Smithtown High School East in St. James, New York. In September of 1977 he entered Worcester Polytechnic Institute in Worcester, Massachusetts, where he participated in research projects with Norton Company and the Worcester Public School System. During his college career he worked at Grumman Aerospace Corporation, Grumman Data Systems Corporation and Data General Corporation. In May of 1981, he graduated with distinction with a Bachelor of Science degree in computer science. Commissioned through the Air Force ROTC program at The College of The Holy Cross in Worcester, Massachusetts, his first assignment was to the Air Force Institute of Technology School of Engineering, Wright-Patterson AFB,

AD-A124 769 ADAPAR: AN ADA RECOGNIZER(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGINEERING W R URE DEC 82 AFIT/GCS/MA/82D-11 2/2

UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/GCS/MA/82D-11	2. GOVT ACCESSION NO. AD-A124969	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* ADAPAR: AN ADA RECOGNIZER		5. TYPE OF REPORT & PERIOD COVERED MS THESIS 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR*(s)* WILLIAM R. URE, 2LT, USAF		8. CONTRACT OR GRANT NUMBER*(s)*
9. PERFORMING ORGANIZATION NAME AND ADDRESS Air Force Institute of Technology Department of Mathematics (AFIT/ENC) WPAFB OH 45433		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE December 1982 13. NUMBER OF PAGES 95
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*		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

Approved for public release: IAW AFR 190-17.

LYNN E. WOLAVER
Dean for Research and Professional Development
Air Force Institute of Technology (ATC)
Wright-Patterson AFB OH 45433

4 JAN 1983

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

Compilers, Recursive Descent Parser, Ada, Syntax Analysis

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

This thesis involved the development of a top down recursive descent Ada recognizer. Basic concepts of compiler theory as they relate to syntax analysis were reviewed. Appropriate syntax diagrams were selected and transformed into program statements using a structured method. The software was developed with attention to software engineering practices. Uses for the recognizer as a programmers tool are discussed. The steps necessary to transform the recognizer into a compiler are discussed. The development of the Ada recognizer was performed on the DECsystem-10 of the AF Avionics Lab at WPAFB Ohio.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*