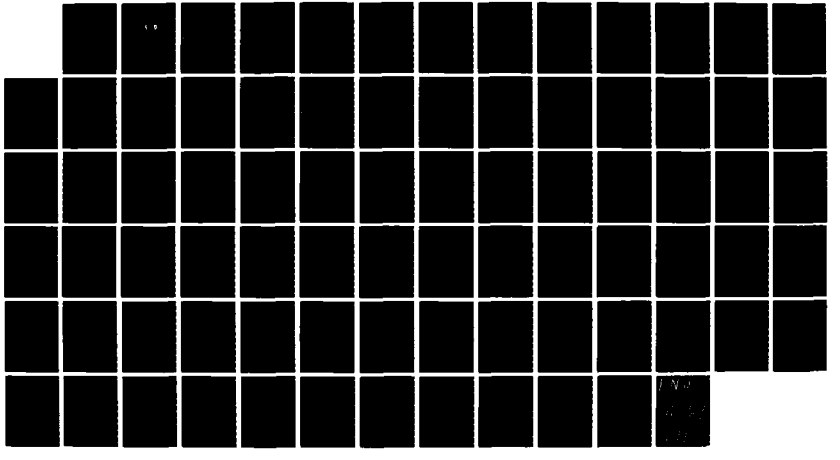
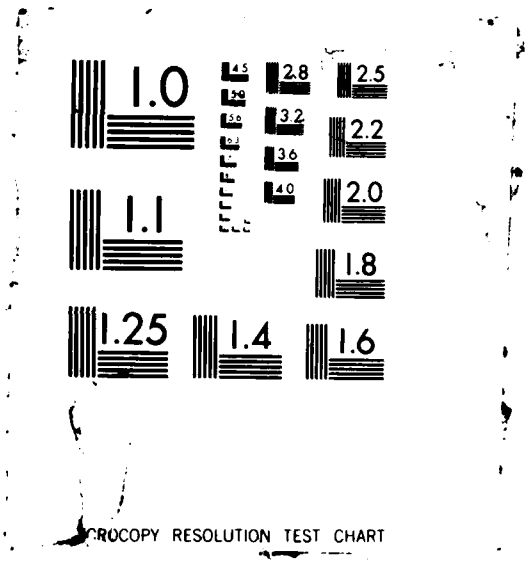
AD-A177 002 A W-GRAMMAR DESCRIPTION FOR ADA(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGINEERING R A FLOWERS DEC 86 AFIT/GCE/ENG/86D-9 1/1

UNCLASSIFIED F/G 9/2 NL




1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
ROCOPY RESOLUTION TEST CHART

AFIT/GCE/ENG/86D

A W-GRAMMAR DESCRIPTION FOR ADA

THESIS

Roy A. Flowers
First Lieutenant, USAF

AFIT/GCE/ENG/86D-9

AFIT/GCE/ENG/86D-9

A W-GRAMMAR DESCRIPTION FOR ADA

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Roy A. Flowers, B.S.E.E

First Lieutenant, USAF

December 1986

Preface

This thesis presents a method for describing the syntax and static semantics of Ada in terms of a W-grammar. The original goal was to create a replacement for the Ada Language Reference Manual. In ~~my~~ the opinion, the W-grammars fall short of this goal since they are less readable than BNF for determining Ada's syntax, and experience shows that programmers most often refer to references to answer questions about syntax.

However, a W-grammar description of Ada is still useful to computer scientists who need more than a simple understanding of the syntax and a rudimentary description of the semantics. A formal semantic definition of Ada is needed by system designers for multiple targets, by compiler designers, and by individuals needing formal correctness proofs of Ada programs.

This thesis could not have been completed without the cooperation and help of others. I would like to thank the members of my thesis committee, Lt Col Seward and Maj Woffinden, for their comments and suggestions which have improved this document 1000% since the first draft. And a special thanks to Capt Jim Howatt, my thesis advisor, who must by now have memorized every word. These people are the special kind of people who make AFIT the institution it is, and I'm proud to have been a part of it. Finally, thanks to my wife, Nancy, who was always there to support me even when I was so busy it seemed I had forgotten her.

Roy A. Flowers

Table of Contents

List of Figures

List of Tables

Abstract

This thesis explores the formal definition of the syntax and static semantics of the Ada programming language. Several notational forms were compared and the particular notational form chosen is a double level grammar called the the W-grammar. W-grammars were first used in the formal definition of Algol 68. Two W-grammars are presented. The first W-grammar is a translation of the modified BNF notation used in the Ada Language Reference Manual, and the second demonstrates the description of Ada's static semantics in W-grammar format.

A W-GRAMMAR DESCRIPTION FOR ADA*

I. Introduction and Literature Review

Background

A common complaint about the Ada language is its complexity. The claim that Ada is complex is borne out by the empirical evidence that the first production quality compilers were not available until over four years after the language specification was complete.

Ada's complexity stems from its sheer size as well as its advanced language constructs (14:4). Ada incorporates most of the modern programming concepts of Algol 68, CLU, Modula, Modula-2, and Pascal in a single programming language. The language structures pioneered by these languages include block structure, strongly typed data structures, separately-compiled code modules, and generic program units. To this already large number of relatively new programming ideas, Ada adds tasking, and the concept of identifier overloading--where a single identifier can have more than one meaning based on its context. It is true that none of the ideas are original with Ada, but attempting to combine so many new ideas into a single language has not been tried since PL/1, and at that time there were far fewer constructs to consolidate.

---

*Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

As modern as the language Ada is, the Ada Language Reference Manual still uses a modified Backus-Naur Form (BNF) meta language notation for the syntax, along with prose descriptions of the semantics (5:1-7 and 1-8). BNF was developed to describe ALGOL, one of the earliest programming languages of all.

An example of the complexity of this methodology for language description can be drawn from the definition of the context_clause in Section 10.1.1 of the Language Reference Manual (5:10-2 and 10-3). The context_clause defines the environment of the current source program and is used to import publicly visible objects from other program units.

The BNF description of context_clause found in the Language Reference Manual is as follows:

```
context_clause ::= {with_clause {use_clause}}
with_clause ::= with unit_simple_name {, unit_simple_name};
```

These two BNF statements are then followed by four paragraphs describing the semantics of the statements. Not until the fourth paragraph is the user told that 'unit_simple_name' must name a previously-compiled library unit. The problem with the Language Reference Manual description is not that BNF is too antiquated for language definition, but that English is too imprecise a language for semantic specifications.

If English were unambiguous there would be no problem with English language specifications for Ada's syntax and semantics, but in fact the ambiguity of English is one of the largest contributors to the proliferation of the legal profession. English is the weakest part of

the Ada language specification, and Ada gurus are often referred to as "Language Lawyers."

The Problem

Syntax vs. Semantics. Compiler developers usually view a programming language from the two aspects of syntax and semantics. Syntax is the mechanical way tokens (the language symbols) are combined to form well-structured language constructs. Semantics is the meaning carried by a language phrase. Semantics can be broken down into two types: static and dynamic.

Static semantics are often hard to distinguish from syntax. As an example consider a simple English sentence with the syntax rule "subject verb object period." The syntax allows such nonsense sentences as "Dog are car." and "House bleeds her." The static semantics of the same sentences require subject-verb agreement between dog and are, and prohibit intransitive verbs like bleeds in this context. In the examples, neither "House bleeds her." nor "Dog are car." would be acceptable to the static semantic rules. Static semantics are often referred to as context-sensitivity.

Even with the syntactic and static semantic rules above, sentences of the form "subject verb object" can be formed which are nonsensical such as "Dog is house." The dynamic semantics, the abstract meanings of the sentence, eliminate the rest of these nonsense "sentences."

For a programming language, the syntax describes valid identifiers and statements, static semantics describe valid blocks and programs, and dynamic semantics describe what happens when a program is run.

Several methods for describing syntax and semantics are discussed later in this chapter.

Purpose of the Study. This study will consider ways to reduce the complexity of the Ada language description by exploring the definitions of Ada in terms of a formal grammar powerful enough to not only reflect a language's syntax, but its semantics as well (both static and dynamic). By using such a grammar to describe both the syntax and semantics formally, we can remove the ambiguity and thus reduce the complexity of English descriptions of the semantics. The type of grammar chosen is the W-grammar (4:46). W-grammars are discussed further in the summary of current knowledge.

Benefits of the Study. Expected benefits of a W-grammar definition for Ada include:

1. A more concise Language Reference Manual. W-grammar definitions have been shown to be smaller than equivalent context-free grammar definitions (4:52-53).

2. A more precise Language Reference Manual. Expressing Ada's semantics formally reduces the inherent ambiguities of English (13:437).

3. Better and more consistent compilers. The Language Reference Manual is the basic compiler requirements document. With a better requirements definition, compiler quality should improve.

4. The possibility of a true Ada compiler-compiler rather than just parser generators (13:437).

5. A better understanding of Ada semantics and a basis for further language improvements.

Each of these expected benefits has ramifications to the Air Force and the Department of Defense as a whole, but of special importance are 2, 3, and 4 which are directly related to the ongoing issues of Ada portability and Ada program reliability.

Scope

This study attempts to begin a formal definition of the Ada programming language syntax and static semantics by creating a W-grammar description of Ada.

Specifically not covered by this study is compiler generation from the W-grammar description. The purpose of this study is to clarify the understanding of Ada syntax and semantics, and although potentially very useful, a compiler generator does not fit within the stated thesis purpose.

Portions of the standard Ada definition will not be covered in this study--Chapters 13 and 14 of the Ada Language Reference Manual, "Representation Clauses and Implementation-Dependent Features" and "Input-Output". The Chapter 13 constructs are inappropriate due to the specific system dependencies, and the chapter 14 constructs are already formally defined (in terms of Ada itself).

Summary of Current Knowledge

This section explores existing alternatives to the methodology used in the Ada Language Reference Manual.

The Chomsky Hierarchy (4:9-20, 7:217-232). In the 1950s, Noam Chomsky defined five classes of phrase structure grammars. This classification scheme has come to be known as the Chomsky Hierarchy.

Phrase structure grammars are composed of four finite sets: the terminal vocabulary V_t, the non-terminal vocabulary V_n, the production set P, and a single distinguished non-terminal symbol S called the root of the language. The five grammar classes within the Chomsky Hierarchy are distinguished by the forms allowed in the set of production rules.

The most restrictive class of languages, which Chomsky did not even name, are the Finite Languages. Production rules in grammars of this class must be of the form S -> x , where S is the designated symbol and x is an element of V_t^*, the Kleene Closure of V_t.

It is easily seen that Finite Languages are simply a finite set of designated strings. The expressive power of such languages is extremely limited.

The next language type, Type 3, the Regular Languages have the Finite Languages as a proper subset. These grammars have production rules of the form A -> aB and A -> b where A and B are elements of V_n and a and b are elements of V_t.

These languages are those accepted by finite automata and have limited use in describing programming languages. Grammars of this type are useful for such things as describing valid language tokens. For example a Type 3 grammar for Ada identifiers is shown in Figure 1.

Type 2 Chomsky Languages are the Context-free Languages. Production rules in context-free grammars have the form A -> n , where A is an element of V_n and n is an element of V^* (V is the union of V_n and V_t).

V_t = {a,b,c,...,x,y,z,A,B,C,...,X,Y,Z,0,1,2,...,7,8,9,_}

V_n = {S,A}

S = {S}

P = {

S -> a, -- an identifier can be a single letter

S -> b, S -> c, ... , S -> x, S -> y, S -> z,

S -> A, S -> B, S -> C, ... , S -> X, S -> Y, S -> Z,

S -> aA, -- an identifier is a letter followed by
-- a string

S -> bA, S -> cA, ..., S -> xA, S -> yA,

S -> zA, S -> AA, S -> BA, S -> CA, ...,

S -> XA, S -> YA, S -> ZA,

A -> a, -- a string may be a single character

A -> b, A -> c, ..., A -> x, A -> y, A -> z,

A -> A, A -> B, A -> C, ..., A -> X, A -> Y, A -> Z,

A -> 0, A -> 1, A -> 2, ..., A -> 7, A -> 8, A -> 9,

A -> _,

A -> aA, -- a string may be multiple characters

A -> bA, A -> cA, ..., A -> xA, A -> yA,

A -> zA, A -> AA, A -> BA, A -> CA, ...,

A -> XA, A -> YA, A -> ZA,

A -> 0A, A -> 1A, A -> 2A, ..., A -> 7A,

A -> 8A, A -> 9A, A -> _A

}

Figure 1. A Type 3 grammar for Ada Identifiers.

These grammars, of which BNF is a member, have sufficient power to describe the syntax of any programming language. In fact, BASIC becomes a context-free language if user-defined functions and arrays are not allowed. Since Type 3 languages are a subset of Type 2 we can describe the Ada identifier in a context-free grammar, but the added power of the context-free grammar makes the definition more concise. Figure 2 is a Type 2 grammar for Ada identifiers.

V_t = {a,b,c,...,x,y,z,A,B,C,...,X,Y,Z,0,1,2,...,7,8,9,_}

V_n = {S,A}

S = {S}

P = {

S -> aA, -- an identifier is a letter followed by
-- a string

S -> bA, S -> cA, ..., S -> xA, S -> yA,

S -> zA, S -> AA, S -> BA, S -> CA, ...,

S -> XA, S -> YA, S -> ZA,

A -> AA -- a string is a sequence of strings

A -> a, -- a string may be a single character

A -> b, A -> c, ..., A -> x, A -> y, A -> z,

A -> A, A -> B, A -> C, ..., A -> X, A -> Y, A -> Z,

A -> 0, A -> 1, A -> 2, ..., A -> 7, A -> 8, A -> 9,

A -> _

}

Figure 2. A Type 2 Grammar for Ada Identifiers.

Type 1 Chomsky Languages are the Context-sensitive Languages. Grammars for these languages have production rules of the form uAn -> uKn , where u and n are elements of V^*, A is an element of V_n, and K is an element of V^+ (the Kleene Closure without the empty string).

Context-sensitive grammars have sufficient power to enforce the data type dependencies of programming languages and are therefore of interest for their ability to describe programming languages. Most useful languages are context-sensitive (7:224).

Here is a trivial Type 1 grammar from Cleaveland and Uzgalis which describes the language $a^n b^n c^n$ where $n > 0$ (4:18):

V_t = {a, b, c}

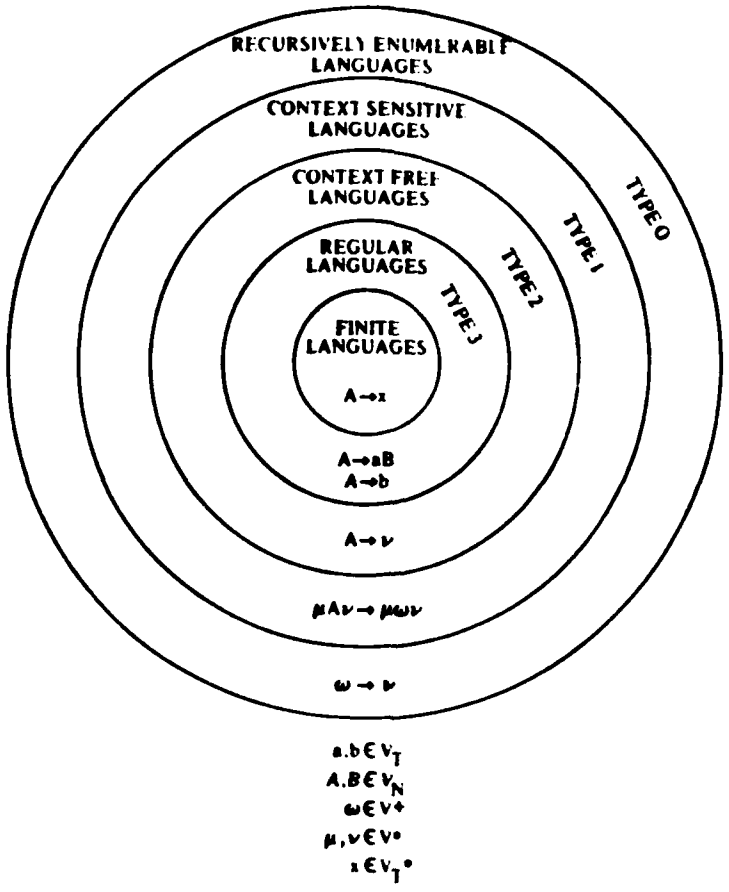
V_n = {S, T, B, C, D}

P = {S -> T, T -> aTBD, T -> abD, DB -> CB, CB -> CD, CD -> BD,
bB -> bb, D -> c}

Type 0 languages, the least restrictive of the phrase-structure languages, are referred to as the Recursively Enumerable languages. These languages are of the form K -> u , where K is an element of V^+ and u is an element of V^*.

There are no trivial examples of recursively-enumerable languages which are not also context-sensitive (4:19), so no example of a strictly Type 0 grammar will be given.

Figure 3 illustrates the relationship between the different Phrase Structure Languages and summarizes the restrictions on the production rules of each type.



(4:20)

Figure 1. The Chomsky Hierarchy

Backus-Naur Form (BNF). Designed by John Backus and Peter Naur, BNF was originally used to define the ALGOL language (2:917; 8:160-162). BNF is able to describe Chomsky Type 2, or context-free languages (7:77-79, 227). Most context-free languages used in language description are similar to BNF; the Ada Language Reference Manual uses a modified form of BNF. The traditional BNF description of an Ada context clause would be:

```
<context clause> ::= <context clause> <with portion> |
  <with portion> ::= <with portion> <with clause> <use portion> |
   <use portion> ::= <use portion> <use clause> |
   <with clause> ::= with <unit simple name list>;
<unit simple name list> ::= <unit simple name> |
                            <unit simple name list>, <unit simple name>
```

Note that this description is significantly longer than the Ada Meta Language where {...} means "zero or more times" and [...] means "zero or one time." Both BNF and the Ada Meta Language are sufficiently powerful to express the syntax of any programming language, but since they are Type 2 grammars, they cannot express the context-sensitive static semantics of programming languages.

BNF and its derivatives are the most widely used vehicles for programming language syntax specification. Because of this, several compiler generation tools are based on BNF. For example, the UNIX (UNIX is a trademark of AT&T) operating system contains a tool called yacc (yacc is an acronym for yet another compiler compiler) which takes a context-free grammar as input and produces a set of tables used to generate a parser (1: YACC manual page).

Semantic Definition Languages. Two common tools for describing static semantics are attribute grammars and the tree grammars such as the Vienna Definition Language.

According to Reps, et al, "an attribute grammar is a context-free grammar extended by attaching attributes to the symbols of the grammar" (9:451). Attribute grammars have been used extensively for

language specification and automated language tools (3; 9; 10), including generation of an Ada compiler (14).

Uhl et al (14) have produced an Ada compiler using an attribute grammar called ALADIN (A Language for Attribute DefINition) and an automated compiler front-end generator called GAG (Generator for Attribute Grammars). The ALADIN code for the syntax portion of the context clause example follows:

```
RULE r_214 :
CONTEXT                 ::= context CONTEXT_ELMS

RULE r_215 :
CONTEXT_ELEMS           ::=

RULE r_216 :
CONTEXT_ELEMS           ::= CONTEXT_ELEM CONTEXT_ELEMS

RULE r_217 :
CONTEXT_ELEM            ::= with NAMES  '%'

RULE r_218 :
CONTEXT_ELEM            ::= USE
```

Vienna Definition Language (VDL) uses tree structures to represent both the syntax and semantics of a programming language (15:1445). It has been used to define several programming languages (4:45; 15:1447). It consists of a syntactic meta language and a semantic meta language. The syntactic meta language groups related syntactic elements in a tree structure called the "abstract syntax." The concrete syntax, the actual order of lexical elements in the language, must still be specified by some other formal method (usually BNF). The semantic meta language is a type of assembly language for a "VDL Machine." It can be analyzed by a VDL interpreter program.

The abstract syntax of the context clause might be expressed as follows:

context_clause = {(<s_1 : with_clause> <s_2 : use_part>),}

use_part = {use_clause,}

with_clause = (<s - op : with> <s_1 : name_list>)

name_list = {(<s_1 : name> <s - op : ,> <s_2 : name_list>), name,}

In addition to attribute grammars and the Vienna Definition Language, compiler semantics have been designated by compiled output for some particular machine architecture. If a compiler produces code which maps into the specified compiled output, then the compiler is correct. The machine architectures used for such language definitions are often similar to the p-machine used by some Pascal compilers, but real machines have been used, and McCarthy actually used LISP as its own definitional language in one of the early papers. (11)

W-Grammars (4). Cleaveland and Uzgalis present an alternative family of grammars called double-level grammars or simply "W-grammars" after their developer Aad Van Wijngaarden who first used them in the formal description of Algol 68. "W-grammars are composed of two context-free grammars: one grammar generates elements that are placed in model production rules, thereby creating rules in the second grammar; the second grammar is used to generate the language." (4:46).

W-grammars are more compact than context-free grammars and have been proven to be Type 0 grammars, general phrase structure grammars. Cleaveland and Uzgalis have shown the power of W-grammars by fully defining the syntax and semantics (both static and dynamic) of a

nontrivial programming language called ASPLE (A Simple Programming Language Example).

The following is a W-grammar expression of the context clause example:

context_clause : with_part repeated.

with_part : with_clause, use_clause repeated.

with_clause : with symbol, unit_simple_name list, SEMICOLON.

This example represents the syntax of the context clause but fails to illustrate the full expressive power of the W-grammar which could also ensure the context-sensitivity of each unit_simple_name (each name must have been previously compiled into the program library). A complete description of W-grammars will be given in Chapter II.

Summary. Most of the work currently being done on semantic description of programming languages is with attribute grammars. This is probably due to the availability of tools such as yacc and GAG (16). W-grammars, developed by Aad Van Wijngaarden hold promise as powerful descriptive tools for combining the syntactic and semantic language descriptions into a single expression.

Approach

After examining the available methods for expressing the semantics of programming languages, W-grammars were chosen. W-grammars have the same expressive ability as attribute grammars or the Vienna Definition Language. However, because W-grammars can include both syntactic and semantic information in the same expression, they make an intuitive link between syntax and semantics which is less evident in the other methods.

The original approach to the design of a W-grammar for Ada was to implement a series of W-grammars for successively larger Ada subsets. It was discovered that subsetting Ada conflicts with the design goal of creating a document which parallels the Ada Language Reference Manual, so the "first cut" grammar is now a rather literal translation of the Ada meta language into W-grammar format. The generation process was invaluable for becoming familiar with W-grammars and in identifying W-grammar "tools" which would aid in the second step.

The second step in the translation process was to choose a basic Ada construct around which a W-grammar could be built--the distinguished non-terminal symbol of a phrase structure grammar. This symbol was then used as the root of a tree upon which the Ada W-grammar could stand. The tree was expanded as many levels as was possible in the time available for the study. The context-sensitive information gleaned from the Ada Language Reference Manual was added to the W-grammar at this time.

Document Overview

The organization of this document parallels the approach discussed in the previous section. Chapter II gives an introduction to W-grammars along with several examples to bring the reader "up to speed" with the terminology used and the structure of W-grammars. Chapter III describes the process used to generate W-grammar A, the direct translation from Ada meta language format. Chapter IV presents W-grammar B which attempts to include static semantic information in the W-grammar. Chapter V presents Ada constructs not covered in W-grammar B, areas worthy of future study, and a summary of the entire thesis.

II. Description of W-grammars

Introduction

As stated in the previous chapter, Aad Van Wijngaarden developed the meta language referred to as the W-grammar, as a descriptive tool to aid in the formal definition of Algol 68. Cleaveland and Uzgalis (4:45-89) give a detailed description of W-grammars as well as many examples. Their nomenclature and syntax for W-grammars differ slightly from the original, and this thesis will follow their description.

Terminology

The character set of W-grammars is separated into three distinct groups: large syntactic marks, small syntactic marks, and separators. Large syntactic marks, commonly called 'capital letters,' are combined to form language tokens called **metanotions**. Small syntactic marks, commonly called 'small letters,' combine to form **protonotions**. As in the case of phrase structure grammars one protonotion is selected as the distinguished symbol (unless otherwise noted, the distinguished symbol is 's'). Metanotions and protonotions are combined to form possibly empty strings called **hypernotions**. Separators, the double colon ('::'), colon (':'), semicolon (';'), and period ('.') carry special meaning in the formation of the W-grammar rules. The double and single colon symbols are assignment symbols; the semicolon separates rule alternatives; and the period is a rule terminator.

A W-grammar is composed of two sets of rules, **metaproductions** and **hyper-rules**, which are combined to form a set of **production rules**.

The metaproductions each define a metanotion. Following is a BNF description of a metaproduction.

<metaproduction> ::= <metanotion> :: <sequence of hypernotions> .
<sequence of hypernotions> ::= <hypernotion> |
<sequence of hypernotions> ; <hypernotion>

The metaproductions form a grammar with metanotions as non-terminals and protonotions for terminals.

The hyper-rules form templates for production rules. Here is the BNF description for a hyper-rule.

<hyper-rule> ::= <hypernotion> : <sequence of hyperalternatives> .
<sequence of hyperalternatives> ::= <hyperalternative> |
<sequence of hyperalternatives> ; <hyperalternative>
<hyperalternative> ::= <hypernotion> |
<hyperalternative> , <hypernotion>

The hyper-rules are a set of rules where the non-terminals are hypernotions and the terminals are protonotions.

The production rules are a possibly infinite set of rules generated by combining the hyper-rules and metanotions whose terminals are protonotions ending in 'symbol.'

<production rule> ::= <protonotion> :: <sequence of prodalternatives> .
<sequence of prodalternatives> ::= <prodalternative> |
<sequence of prodalternatives> ; <prodalternative>
<prodalternative> ::= <protonotion> | <prodalternative> , <protonotion>

An Analogy

The nomenclature and syntax of W-grammars can be overwhelming the first time they are encountered. A way to simplify the W-grammar concept is to use your existing knowledge by drawing an analogy between W-grammars and programming languages.

The metaproductions form the declarative part of a W-grammar. Each individual metaproduction is like a variable declaration for a metanotion which acts as a W-grammar variable. And the hypernotions right of the double colon define the type of the variable.

The hyper-rules form the procedural portion of a W-grammar. Each hyper-rule is like a function which takes metanotions as inputs and returns production rules as outputs.

At a global level, W-grammars are like programs which continuously produce members of the set of strings referred to as the language of the W-grammar.

Uniform Replacement Rule

The hyper-rules and metanotions are combined in a manner called the Uniform Replacement Rule.

As stated above, the hyper-rules form a pattern for the production rules. Production rules are generated from hyper-rules by replacing any metanotion by a terminal metaproduction of that metanotion. If any metanotion occurs more than once in the hyper-rule, each occurrence of the metanotion is replaced by the same terminal metaproduction.

An extension to the W-grammar which adds no power, but increases brevity and readability is to treat metanotions which are the same except for possibly a digit as their right-most character, as if they

share the same defining metaproduction, but as distinct metanotions with regards to the Uniform Replacement Rule.

In the programming language analogy, the Uniform Replacement Rule can be viewed as a rule for binding variables to values.

A Finite Example

Following is an example of a W-grammar with a finite number of production rules to illustrate the Uniform Replacement Rule.

Metaproductions

ALPHA :: a; b.

Hyper-rules

s : t; u.

t : letter ALPHA symbol, letter ALPHA symbol.

u : letter ALPHA1 symbol, letter ALPHA2 symbol.

After applying the Uniform Replacement Rule to the hyper-rules we end up with the following set of production rules.

s : t; u.

t : letter a symbol, letter a symbol.

t : letter b symbol, letter b symbol.

u : letter a symbol, letter a symbol.

u : letter a symbol, letter b symbol.

u : letter b symbol, letter a symbol.

u : letter b symbol, letter b symbol.

Note the difference in the hyper-rules for t and u and how the Uniform Replacement Rule affected the output.

A W-grammar for an Infinite Language

The following W-grammar describes the language of all strings consisting of zeros and ones with an odd number of ones.

Metaproductions

ZEROETY :: EMPTY; ZEROETY zero symbol.

EMPTY :: .

Hyper-rules

s : ZEROETY1, one symbol, ZEROETY2;

ZEROETY1, one symbol, ZEROETY2, one symbol, s.

Replacing **ZEROETY1** and **ZEROETY2** by **EMPTY** yields:

s : one symbol; one symbol, one symbol, s.

which covers the cases $(11)^*1$. Replacing **ZEROETY1** by **zero symbol** and replacing **ZEROETY2** by **EMPTY** yields:

s : zero symbol, one symbol; zero symbol, one symbol, one symbol, s.

which covers the cases $(011)^*01$. Combination of the two production rules yields $(011 + 11)^*(01 + 1)$. The reader should generate additional production rules until he is satisfied the W-grammar covers all the possible cases.

Notice how the metanotion **EMPTY** is used to make the grammar more concise. Note also that two metanotions and one hyper-rule can represent an infinite number of production rules.

W-grammar Summary

This chapter presented an introduction to a descriptive language developed by Aad Van Wijngaarden called the W-grammar.

W-grammars consist of a series of metanotion definitions called metaproductions and one or more production rule templates called hyper-rules. The production rules, whose terminals end in **symbol**, are formed by replacing the metanotions in the hyper-rules according to the Uniform Replacement Rule, so that a finite number of metaproductions and hyper-rules can generate a possibly infinite number of production rules.

Now that the preliminaries are complete, we can begin to create the Ada W-grammar.

III. W-grammar A

The Initial Translation

The first iteration of the Ada grammar, W-grammar A, is a literal translation of the Ada meta language definitions from the Ada Language Reference Manual. A literal translation was chosen as a way to become more acquainted with W-grammars, and as a way to capitalize on the existing formal syntax definition, as well as a way to insure the complete language is addressed.

W-grammar A appears in Appendix A. The chapter and section numbers in the appendix relate to the section number of the corresponding context-free grammar rule in the Ada Language Reference Manual.

Metaproductions

CHAR :: a; b; c; d; e; f; g; h; i; j; k; l; m; n; o; p; q; r; s; t; u; v; w; x; y; z; _.

NOTION :: CHAR; NOTION CHAR.

EMPTY :: .

Hyper-rules

NOTION option : NOTION; EMPTY.

NOTION repeated : NOTION repeated, NOTION; NOTION; EMPTY.

NOTION list : NOTION; NOTION list, comma symbol, NOTION.

NOTION sequence : NOTION; NOTION sequence, semicolon symbol, NOTION.

NOTION pack : left parenthesis symbol, NOTION list, right parenthesis symbol.

NOTION group : left parenthesis symbol, NOTION sequence, right parenthesis symbol.

Figure 4. Some Hypernotions Used in W-grammar A.

The W-grammar Tools

A series of metaproductions and hyper-rules influenced by Cleaveland and Uzgalis (4:53-62) were developed to aid in the translation process. These metanotions, shown in Figure 4, are designed to replace the bracket and brace extensions to BNF used in the Ada Language Reference Manual.

The underscore character was included for use in W-grammar A so the identifier names used in the Language Reference manual could be used in the W-grammar also. A specific underscore should be considered as a large or small syntactic mark depending on the case of the word it is in. Underscores not part of a word are considered to be small syntactic marks.

The **option** and **repeated** constructs replace the square bracket and brace symbols of the Ada meta language. A W-grammar **option** clause causes the clause suffix to be considered optional since the protonotion NOTION **option** may be replaced by either NOTION or EMPTY. The **repeated** clause works similarly in that the clause suffix represented by NOTION may occur zero (EMPTY), once, or several times.

Other W-grammar constructs developed include the **list**, a series of NOTIONs separated by commas, and the **sequence** which is similarly separated by semicolons. In addition, **pack** and **group** are respectively a **list** and a **sequence** surrounded by parentheses.

An Example

The meta language to W-grammar translation which generated W-grammar A was a simple mechanical process. It consisted of replacing the BNF assignment symbol '::=' by ':', separating the language notions

by commas, and inserting the option and repeated protonotions in place of brackets and braces in the Ada meta language. A representative Ada construct, the task declaration, is converted from the Ada meta language to W-grammar A form below.

The original Ada BNF representation (5:9-2) is:

```
task_declaration ::= task_specification;

task_specification ::=
   task  [type]  identifier  [is
      {entry_declaration}
      {representation_clause}
   end  [task_simple_name]]

task_body ::=
    task body task_simple_name is
        [ declarative_part]
    begin
         sequence_of_statements
   [ exception
         exception_handler
       { exception_handler}]
    end  [task_simple_name];
```

Translating the first line simply requires changing the assignment symbol and separating the right-hand notions by commas. The metanotion SEMICOLON is used to represent the statement terminator. The metaproduction for SEMICOLON is

SEMICOLON :: semicolon symbol.

With these changes the first expression becomes:

task_declaration : task_specification, SEMICOLON.

The W-grammar expression for task_specification reveals another convention adopted for the Ada W-grammars: Ada reserved words are

treated as atomic symbols. In the W-grammar, a reserved word is represented by that word followed by the production rule terminal designator symbol.

An English description of the task_specification description given above might say:

> The task_specification begins with the reserved word **task** and the optional reserved word **type** followed by an identifier. The remainder, which is also optional, consists of the reserved word **is** followed by zero or more entry_declarations immediately followed by zero or more representation_clauses, and terminated by the reserved word **end** followed by an optional task_simple_name.

In order to increase readability in the W-grammar, the large optional portion beginning with is is moved to a separate definition called is_part. The W-grammar definition of task_specification is:

```
task_specification : TASK, TYPE option, identifier, is_part option.

is_part : IS, entry_declaration option, representation_clause
          option, END, task_simple_name option, SEMICOLON.
```

TASK, TYPE, IS and END are metanotions representing the terminal protonotions task symbol, type symbol, is symbol, and end symbol.

Finally, the W-grammar translation for task_body is:

```
task_body : TASK, BODY, task_simple_name, IS, declarative_part
            option, BEGIN, sequence_of_statements, exception_part
            option, END, task_simple_name option, SEMICOLON.

exception_part : exception_handler, exception_handler option.
```

exception_part, like is_part, was created to increase readability.

Italicized Names

Although W-grammar fully describes Ada's syntax, it fails to reflect any semantic information. An indication of this is the number of names used which have italicized parts in the Ada Language Reference Manual, where "the italicized part is intended to reflect some semantic information" (5:1-8).

The italicized names found in the Ada Language Reference Manual are listed in Table I. These items are used in the Ada Language Reference Manual and W-grammar A to represent specific declared items whose use in the program depend on the actual context. These constructs must be further defined in the W-grammar to reflect their context-sensitivity.

Table I. Italicized Terms in the Ada Language Reference Manual.

Section	Term
2.8	argument_identifier
3.2	universal_static_expression
3.3.2	type_name
	subtype_name
3.5	range_attribute
3.5.7	static_simple_expression
3.6	component_subtype_indication
	discrete_subtype_indication
3.7.2	discriminant_simple_name
3.7.3	component_simple_name
5.2	variable_name
5.3	boolean_expression
5.5	loop_simple_name
5.6	block_simple_name
5.7	label_name
6.4	procedure_name
	function_name
	parameter_simple_name
7.1	package_simple_name
8.4	package_name
8.5	object_name
	exception_name

Table I. Italicized Terms in the Ada Language Reference Manual (cont.).

Section	Term
	subprogram_or_entry_name
9.1	task_simple_name
9.5	entry_name
	entry_simple_name
10.1.1	unit_simple_name
10.2	parent_unit_name
12.3	generic_package_name
	generic_procedure_name
	generic_function_name
	subprogram_name
13.3	type_simple_name
	component_name
	static_range
13.8	record_aggregate

The items in this list represent identifiers or expressions. The static semantics of each of these is similar in that each depends on a previous declaration or on the inherent type of a constant.

A typical example from this group is procedure_name. The static semantics require a procedure to be defined before it can be called. A definition of procedure_name reflecting this requirement can be produced in a W-grammar:

procedure_name :: IDENTIFIER, where IDENTIFIER is defined
    with type PROCEDURE.

The static semantics problem is in defining the context-sensitive phrase **where NOTION1 is defined with type NOTION2** in such a way that the phrase can be parsed by the W-grammar. The static semantics problem is addressed in the second W-grammar found in Appendix B and discussed in Chapter IV.

IV. W-grammar B

Introduction

W-grammar B found in Appendix B contains a partial Ada grammar displaying a method for expressing Ada's context-sensitivity in W-grammar form. This W-grammar is a complete syntactic and static semantic description of an Ada compilation down to the package/subprogram level. It successfully demonstrates the ability of W-grammars to not only express Ada's syntax but its static semantics as well. This chapter describes the development and structure of W-grammar B.

W-grammar B in Relation to W-grammar A

The original motivation for creating W-grammar A was to build a framework to which static semantic restrictions could be added to create a context-sensitive W-grammar. Early in the development of W-grammar B it became apparent that the nature of a W-grammar which included semantic information was very different from the purely syntactic W-grammar A. Therefore, a better grammar would result from a whole new development rather than from trying to coerce the W-grammar A design into a context-sensitive form.

W-grammar B utilizes the knowledge gained while designing W-grammar A about W-grammar design, as well as how to make Ada constructs fit into W-grammar descriptions. In addition, most of the W-grammar tools shown in Figure 2 were used in designing W-grammar B.

The Ada Program Concept

W-grammars, BNF, and phrase structure grammars require a concept called the "distinguished symbol" or "the root of the language" (4:9-10). For most languages, the language root is a symbol representing a compiled program. Separate compilation is handled by ignoring the static semantics of separately-compiled subroutines until link time. Ada is different from most languages in that it requires the semantics of separately compiled program units to be checked through the use of the library. Therefore there is really no Ada equivalent of a stand-alone program: every Ada program is compiled in the context of the existing Ada program library.

The clause **LIBRARY compilation** from hyper-rule B.34 is a rough equivalent for an Ada language root: the "distinguished non-terminal symbol" of BNF or a phrase structure grammar. This clause is a hypernotion (containing a metanotion, a W-grammar 'variable') rather than a protonotion (a W-grammar 'constant'). The true root of the language must be a protonotion since it must be a non-terminal of a production rule. The true Ada root would be something like **environment** which would include all the predefined packages as well as all library units which have ever been compiled into that particular Ada program library.

LIBRARY compilation clearly infers the semantic fact that all compilations must take place in the context of the program library.

LIBRARY

To handle the context-sensitivity of an Ada compilation the **LIBRARY** clause was developed. **LIBRARY** is defined in Metaproduction

B.K as a series of **library entry** clauses separated by **nextentry** and terminated by **endlibrary**. LIBRARY serves as a type of symbol table of previously compiled library units as well as the Ada predefined environment. It carries all the syntactic and semantic information necessary to describe the current compilation environment. The clause **library entry** is defined by Hyper-rule B.14 as a **NAME** followed by a DESCRIPTION; there is one **library entry** for each previously defined library unit.

A **NAME**, Metaproduction B.J, is a series of characters each separated by the token **name**. The **NAME** portion of a **library entry** represents the identifier associated with the **library entry**. Using the **name** tokens simplifies the process of locating the particular **library entry** within LIBRARY.

DESCRIPTION is the part of a **library entry** which lists the attributes of the object represented by the **library entry**. DESCRIPTION is defined by Metaproduction B.R.

To illustrate how the **LIBRARY** concept works in W-grammar B consider the following hyper-rule for construction of a **foo construct**.

```
LIBRARY foo construct : foo symbol, IDENTIFIER print, SEMICOLON,
                        where ID is in LIBRARY
                           and ID describes IDENTIFIER.
```

This hyper-rule describes something which could be described in BNF by

```
<foo construct> ::= foo <identifier> ;
```

with the additional constraint that the **IDENTIFIER** be declared in

LIBRARY. This is clearly a context-sensitive constraint. Now we will analyze how this rule is used to parse a language string.

Comparing the BNF and W-grammar rules, one can easily see that the first three hypernotions on the right side of the colon describe the syntax of the foo construct. Therefore the **where** portion of the description must describe the static semantics.

How does it work? Remember from Chapter II that the terminal protonotions of the production rules must end in **symbol**. The **where** clause is formed so that it parses into an empty string or a protonotion not ending in **symbol**. If the **where** clause goes away (parses into an empty string) then the static semantics are correct, but if it reduces to any other protonotion, it won't be a terminal and therefore the grammar won't be able to parse the candidate string.

Consider the statement

foo junk;

This statement should parse if and only if junk is in the library.

Consider the case where the statement is correct:

1. If the statement is correct, then junk must be in the library.
2. If junk is in the library, then **LIBRARY**, which reflects the contents of the library, must be of the form "nextentry...nextentrynamejnameunamennamekkindof...endlibrary" where the ellipses contain other characters with which we are not concerned.
3. If junk is in the library, then the **library entry** for junk is of the form

"namejnameunamennamekkindof..." which is also a substring of LIBRARY.

Now substitute the appropriate strings into the hyper-rule (don't forget the Uniform Replacement Rule which applies to the multiple instances of **LIBRARY**, **IDENTIFIER**, and **ID**). The original rule is:

```
LIBRARY foo construct : foo symbol, IDENTIFIER, SEMICOLON,
                        where ID is in LIBRARY
                           and ID describes IDENTIFIER.
```

Where **ID** is defined by metaproduction B.L:

```
ID :: library entry.
```

Substituting for **LIBRARY** and **ID** gives us

```
nextentry...nextentrynamejnameunamennamekkindof...endlibrary
     foo construct : foo symbol, IDENTIFIER, semicolon symbol,
                     where namejnameunamennamekkindof... is in
          nextentry...nextentrynamejnameunamennamekkindof...endlibrary
          and namejnameunamennamekkindof... describes IDENTIFIER.
```

Applying metanotion B.I,

```
IDENTIFIER :: LETTER; IDENTIFIER, UNDERSCORE option, LETTER option,
              DIGIT option.
```

we can obtain

```
IDENTIFIER :: junk.
```

Note that the Uniform Replacement Rule does not apply to metaproductions.

Substituting again, we get

```
nextentry...nextentrynamejnameunamennamekkindof...endlibrary
     foo construct : foo symbol, junk print, semicolon symbol,
                     where namejnameunamennamekkindof... is in
          nextentry...nextentrynamejnameunamennamekkindof...endlibrary
          and namejnameunamennamekkindof... describes junk.
```

Then hyper-rule B.14 can be used to form the production rule

```
    junk print : letter j symbol, letter u symbol, letter n symbol,
                 letter k symbol.
```

Substituting once again gives us

```
nextentry...nextentrynamejnameunamennamekkindof...endlibrary
     foo construct : foo symbol, letter j symbol, letter u symbol,
                     letter n symbol, letter k symbol, semicolon symbol,
                     where namejnameunamennamekkindof... is in
          nextentry...nextentrynamejnameunamennamekkindof...endlibrary
          and namejnameunamennamekkindof... describes junk.
```

which gives us the complete syntactic structure.

Now for the semantics...

The last protonotion is of the form of hyper-rule B.23:

```
     where NOTETY1 and NOTETY2 : where NOTETY1; where NOTETY2.
```

Substituting 'namejnameunamennamekkindof... is in

nextentry...nextentrynamejnameunamennamekkindof...endlibrary' for

NOTETY1 and 'namejnameunamennamekkindof... describes junk' for NOTETY2

we get:

where namejnameunamennamekkindof... is in
nextentry...nextentrynamejnameunamennamekkindof...endlibrary and
namejnameunamennamekkindof... describes junk :
where namejnameunamennamekkindof... is in
nextentry...nextentrynamejnameunamennamekkindof...endlibrary,
where namejnameunamennamekkindof... describes junk.

Substituting once again into the original equation:

nextentry...nextentrynamejnameunamennamekkindof...endlibrary
foo construct : foo symbol, letter j symbol, letter u symbol,
letter n symbol, letter k symbol, semicolon symbol,
where namejnameunamennamekkindof... is in
nextentry...nextentrynamejnameunamennamekkindof...endlibrary,
where namejnameunamennamekkindof... describes junk.

Now the **where...and** clause has become two simpler **where** clauses.

Working right to left we find that the last **where** clause matches B.29 and that the other **where** clause matches B.19. The B.19 resolution is trivial, but the B.29 resolution requires several applications of hyper-rule B.29, and for the sake of saving space, the completion of this exercise is left to the reader.

The final production rule is

nextentry...nextentrynamejnameunamennamekkindof...endlibrary
foo construct : foo symbol, letter j symbol, letter u symbol,
letter n symbol, letter k symbol, semicolon symbol.

which maps into the syntax

foo junk;

and brings us back where we began. If junk had not have been in the **LIBRARY** structure, the **where** clauses would not have resolved to **EMPTY** and the statement 'foo junk;' would have been impossible to parse and hence not part of the language.

The Development of W-grammar B

After the language root **LIBRARY compilation** was defined, W-grammar B was developed in a tree-like manner; Figure 5 represents the development strategy followed. Hyper-rule B.34 is equivalent to the similar Ada meta language statement in Section 10.1 of the Ada Language Reference Manual (5:10-1).

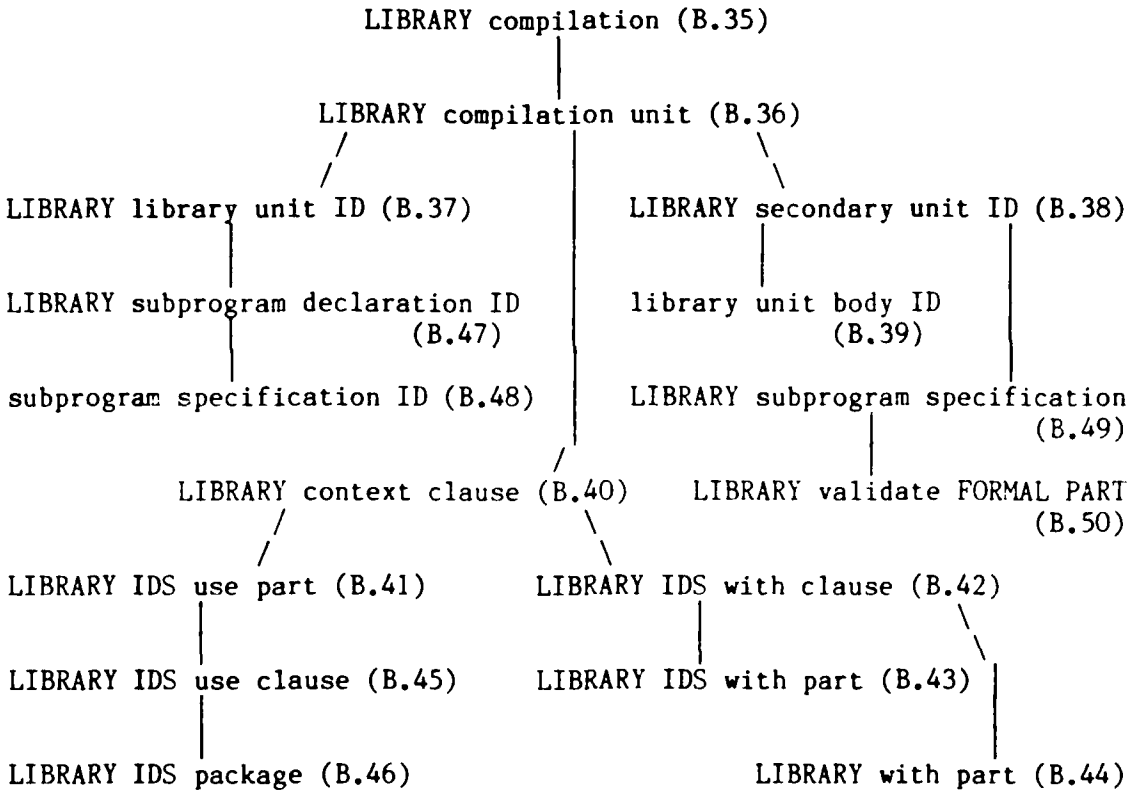


Figure 3. W-grammar B Development.

New Tools in W-grammar B. Before describing the basic constructs which appear in Figure 5, we will examine the peripheral hyper-rules necessary to add static semantics to the Ada W-grammar.

Rules B.17 through B.34 (beginning with the symbol where) form a set of boolean expressions which give the W-grammar the expressive power to selectively parse only expressions which are correct by the context-sensitive rules of W-grammar B. These rules work because W-grammar production rules terminate with clauses ending with the symbol **symbol.** If the **where...** clause is true it will parse to **EMPTY** and cease to exist; if it is false it will not parse to **EMPTY** or a clause ending in **symbol** resulting in a 'dead end' parse tree--an invalid language construct.

Rules B.18 through B.22 handle equality of strings. Note that proving things not equal is harder than proving them equal. It is easy to search one string for the presence of another string in a W-grammar using a structure like **NOTETY1 STRING NOTETY2.** But in order to search for the absence of a string you must search the larger string by "peeling it off" one character at a time.

Rules B.23 through B.26 check for membership in a string. These clauses are the most useful for adding context-sensitivity since the static semantic information is kept in long strings called **LIBRARY** and **IDS.**

Rules B.27 and B.28 are the boolean operators and and or. They do not behave exactly like their mathematical counterparts and care must be taken when using them--especially when mixing **and** and **or.** This is because they are string manipulators and have no precedence order.

Rules B.29 through B.34 are used to check semantic information kept in an **ID.** B.29 and B.30 are used to check whether a particular **ID**

describes a particular **IDENTIFIER**, and B.31 through B.34 check the attributes of the ID.

The Ada Subset. The rest of the rules, B.34 through B.54, describe the Ada subset implemented in W-grammar B. Due to time constraints, only a subset of Ada was considered. The actual Ada subset is small and nonfunctional having only 16 semantic constructs, but it is sufficiently large to demonstrate the context-sensitivity of the W-grammar. The Ada syntactic units included are listed in Table II.

Table II. Ada Constructs Included in W-grammar B.

Ada Language Reference Manual Section	Ada Constructs
2.1	graphic_character basic_graphic_character basic_character
2.3	identifier
3.9	proper_body
6.1	subprogram_declaration subprogram-specification formal_part parameter_specification mode
8.4	use_clause
10.1	compilation compilation unit library_unit secondary_unit library_unit_body
10.1.1	context_clause with_clause
10.2	subunit

As stated above, rule B.35 describes the clause **LIBRARY compilation** which is used as the root of the language. It is defined as a series of compilation units.

Rule B.36 expands a compilation unit as a context clause followed by either a library unit or a secondary unit. A library unit, rule B.37, is a separate program unit which upon compilation becomes a part of the program library. Subprogram bodies can be either library units or subunits depending on whether the subprogram has been declared to the library. The correct type of subprogram body is identified with B.37 by the clause **where ID is not in LIBRARY.**

B.38 is the definition of a subunit. Subunits must already have been defined in the library at compilation time; these semantics are enforced by the clause **where ID is in LIBRARY.**

Rule B.39 describes a library unit body as either a subprogram body or a package body.

A context clause is described in rule B.40 as a with clause followed by a repeated use part, and rule B.41 describes the use part as the symbol use followed by repeated use clauses. The with clause, rule B.42, is with followed by a with part ending with a semicolon.

Rules B.43 and B.44 describe the with part. The rules require that all the IDs in IDS and only those IDs are referenced. These two rules form a recursive definition with B.44 forming the escape clause.

B.45 defines the use clause, and B.46 enforces the rule that the packages used in the use clause were referenced in the previous with clause.

Rules B.47 and B.48 define the subprogram declaration referred to by B.37, rules B.49 and B.50 define subunits as referred to in B.38.

Summary

This chapter presented the development of a partial W-grammar for Ada which can handle syntactic and static semantic requirements of compilation units through the declaration of the top level program units. Static semantics were handled through the use of a symbol-table-like **LIBRARY** construct which represents the programming library environment, and selective parsing of valid expressions was done through **where...** clauses which parse only when presented with a true expression.

V. Conclusion

This chapter covers the possibility of representing several other Ada syntactical constructs in W-grammar form, topics which appear worthy of future study, and finally, a summary of the entire thesis document.

Ada Constructs Not Covered in W-grammar B

Ada constructs considered are those originally thought hard to represent in W-grammar form due to the advanced nature of their semantics. The increased familiarity with W-grammars and with the description of Ada static semantics in W-grammars gained while designing the two W-grammars described in this document has helped the author see simple (though not trivial) implementation solutions for the static semantics for each construct.

These solutions were not addressed in W-grammar B due to time constraints. The specific Ada constructs addressed are generic units, tasks, and overloading.

Generics. Generic units should not be difficult to incorporate in a W-grammar. The generic definition could be added to the library like any other definition and the instantiation would simply add the instantiated name along with any parameters to the library as well.

As an example, a possible W-grammar hyper-rule for generic instantiation might be.

```
LIBRARY generic instantiation :
     NOTION1, IDENTIFIER1, IS, NEW, IDENTIFIER2, ID actual part,
     where ID is in LIBRARY,
     where ID describes IDENTIFIER1,
     where ID kind generic NOTION1.
```

This rule requires the definition of **NOTION actual part** which is analogous to generic_actual_part in the Language Reference Manual, but all the other clauses in this definition have previously been defined in W-grammar B.

Tasks. Tasks are simple to incorporate in W-grammar B since they have static semantics very similar to the other types of program units.

Tasks may cause considerable problems for anyone attempting to express the dynamic semantics though, since Ada task-related statements (notably the **select** statement) can be nondeterministic. Nondeterminism is expressed in W-grammar statements by multiple hypernotions separated by semicolons on the right hand side of a W-grammar rule (nondeterminism in a grammar requires a parser to make a choice between two or more options).

Overloading. Overloading can be easily handled by structuring library searches so they search in reverse order of library growth. By searching in reverse order and requiring that parameter types match, the correct object will be found when searching for an overloaded name. Overloading is similar to defining local variables with the same names as global variables.

For an example consider the case where the "+" operator is overloaded for two integers. The earliest library entry for "+" with two integer parameters will be the predefined entry from **package STANDARD**, and any user definition of "+" for two integers will occur later in the symbol table (toward the right end of the string **LIBRARY**). If the **LIBRARY** search routine searches from right to

left, it will encounter the user's definition first, use it, and cease searching with the correct entry for "+".

Areas for Further Study

This thesis has uncovered several promising areas for future investigation related to Ada syntax and semantics as well as the syntax and semantics of programming languages in general.

The first area of interest of course, is the completion of W-grammar B. Such a continuation of this work would test the assertions made earlier in this chapter, as well as possibly lead to the benefits addressed in chapter I. In addition, a complete Ada W-grammar including static semantics would be useful as an academic case study for the topics below.

More study of W-grammars is necessary. While preparing the W-grammars in this study the author noted that the readability, and therefore the useability, of a W-grammar depends heavily on stylistic decisions made by the W-grammar designer. Research into which W-grammar styles are most understandable, and guidelines for designing understandable W-grammars is necessary before W-grammars can be used as a medium expressing military standards.

W-grammars also seem to express certain constructs better than others as in the case of the W-grammar B hypernotions "where NOTION is in NOTETY" and "where NOTION is not in NOTETY." Defining the first expression requires a single one-line hypernotion, while defining the second requires 6 hypernotions for a total of 13 lines of W-grammar code. Research in this area could reveal the type of programming languages best suited to W-grammar definition.

More study is needed in the area of formal expression for semantics--especially in the area of an expression medium. W-grammars may prove too cumbersome for use in situations where the individual grammatical symbols must be manipulated. This thesis proves the possibility of describing Ada's static semantics formally, but some other method such as axiomatic definition might produce a more useful form for such uses as formal correctness proofs.

Thesis Summary

Ada is a modern programming language reflecting the most recent knowledge about software engineering. But the same attributes which make Ada a good choice for a programming language, namely its modern language constructs which promote program reliability and maintainability, also make Ada a large and complex language.

The formal definition of Ada is in terms of a modified BNF grammar describing its syntax augmented by a prose description of its semantics. This thesis explores a method for improving the formal definition of Ada by formally defining the syntax and static semantics of Ada in terms of a W-grammar.

A W-grammar is composed of two context-free grammars which are joined to form BNF-type production rules by a method known as the Uniform Replacement Rule

Two Ada W-grammars are given in the Appendices. W-grammar A is a translation from the Ada meta language; it demonstrates the ability of a W-grammar to express Ada's full syntax.

W-grammar B presents not only the syntax, but the static semantics as well. This W-grammar does not present the complete Ada language, but

due to time constraints, it simply presents a portion of the language sufficiently large to demonstrate the ability of a W-grammar to describe Ada's static semantics.

This thesis demonstrates the ability of W-grammars to describe the syntax and static semantics of Ada. It is expected that formal definition of Ada's semantics, both static and dynamic, will add to the language's portability and reliability. The results presented here help support these goals.

Appendix A: W-grammar A

2.1 Character Set

```
graphic_character : basic_graphic_character; lower_case symbol;
     other_special_character symbol.

basic_graphic_character : upper_case symbol; digit symbol;
     special_character symbol; space_character.

basic_character : basic_graphic_character; format_effector.

upper_case : capital letter.

digit : zero; one; two; three; four; five; six; seven; eight; nine.

special_character : quotation; sharp; ampersand; apostrophe;
     left parenthesis; right parenthesis; star; plus; comma; hyphen;
     dot; slash; colon; semicolon; less than; equal; greater than;
     underline; vertical bar.

lower_case : letter letter.

other_special_character : exclamation mark; dollar; percent; question
     mark; commercial at; left square bracket; back-slash; right square
     bracket; circumflex; grave accent; left brace; right brace; tilde.

letter : a; b; c; d; e; f; g; h; i; j; k; l; m; n; o; p; q; r; s; t; u;
     v; w; x; y; z.

COMMA :: comma symbol.
COLON :: colon symbol.
SEMICOLON :: semicolon symbol.
LPAREN :: left parenthesis symbol.
RPAREN :: right parenthesis symbol.
PLUS :: plus symbol.
HYPHEN :: hyphen symbol.
STAR :: star symbol.
SLASH :: slash symbol.
GT :: greater then symbol.
LT :: less than symbol.
EQUAL :: equal symbol.
UNDERLINE :: underline symbol.
APOSTROPHE :: apostrophe symbol.
```

2.2 Lexical Elements, Separators, and Delimiters

delimiter : simple_delimiter symbol; compound_delimiter symbol.

simple_delimiter : ampersand; apostrophe; left parenthesis; right parenthesis; star; plus; comma; hyphen; dot; slash; colon; semicolon; less than; equal; greater than; vertical bar.

compound_delimiter : arrow; double dot; double star; assignment; inequality; greater than or equal; less than or equal; left label bracket; right label bracket; box.

TABLE I. Compound Delimiters in W-grammar A.

grammatical symbol	delimiter
arrow symbol	=>
double dot symbol	..
double star symbol	**
assignment symbol	:=
inequality symbol	/=
greater than or equal symbol	>=
less than or equal symbol	<=
left label bracket symbol	<<
right label bracket symbol	>>
box symbol	<>

ARROW :: arrow symbol.
DDOT :: double dot symbol.
DSTAR :: double star symbol.
ASSIGN :: assignment symbol.
INEQUALITY :: inequality symbol.
GE :: greater than or equal symbol.
LE :: less than or equal symbol.
LLABEL :: left label bracket symbol.
RLABEL :: right label bracket symbol.
BOX :: box symbol.

2.3 Identifiers

idchars : underline option, letter_or_digit symbol.

a_letter : lower_case; upper_case.

identifier : a_letter symbol, idchars option.

letter_or_digit : a_letter; digit.

2.4 Numeric Literals

numeric_literal : decimal_literal; based_literal.

2.4.1 Decimal Literals

fraction : dot symbol, integer.

intchars : UNDERLINE option, digit symbol.

decimal_literal : integer, fraction option, exponent option.

integer : digit symbol, intchars option.

exponent : capital e symbol, sign, integer.

sign : PLUS; HYPHEN; EMPTY.

2.4.2 Based Literals

based_literal : base, sharp symbol, based_integer, bfraction option,
sharp symbol, exponent option.

bfraction : dot symbol, based_integer.

base : integer.

based_integer : extended_digit symbol, bchars option.

extended_digit : a_letter; digit.

bchars : underline OPTION, extended_digit symbol.

2.5 Character Literals

character_literal :: APOSTROPHE, graphic_character, APOSTROPHE.

2.6 String Literals

string_literal :: quotation symbol, graphic_character option, quotation
symbol.

2.8 Pragmas

pragma :: pragma symbol, identifier, pragma_argument option.

pragma_argument :: LPAREN, argument_association list, RPAREN.

argument_association :: argid option, name; argid option, expression.

argid :: argument_identifier, ARROW.

2.9 Reserved Words

In the W-Grammar reserved words will be identified by the word followed by "symbol".

AND :: and symbol.
AT :: at symbol.
BEGIN :: begin symbol.
BODY :: body symbol.
CONSTANT :: constant symbol.
ELSE :: else symbol.
END :: end symbol.
FOR :: for symbol.
IF :: if symbol.
IN :: in symbol.
IS :: is symbol.
LIMITED :: limited symbol.
NEW :: new symbol.
NOT :: not symbol.
NULL :: null symbol.
OF :: of symbol.
OR :: or symbol.
OUT :: out symbol.
PACKAGE :: package symbol.
RECORD :: record symbol.
RENAMES :: renames symbol.
SELECT :: select symbol.
TASK :: task symbol.
THEN :: then symbol.
TYPE :: type symbol.
USE :: use symbol.
XOR :: xor symbol.

3.1 Declarations

basic_declaration : object_declaration; number_declaration;
type_declaration; subtype_declaration; subprogram_declaration;
package_declaration; task_declaration; generic_declaration;
exception_declaration; generic_instantiation; renaming_declaration;
deferred_constant_declaration.

3.2 Objects and Named Numbers

object_declaration : identifier list, COLON, constant option,
subtype_indication, assignment option, SEMICOLON; identifier list,
COLON, constant option, constrained_array_definition, assignment
option, semicolon.

assignment : ASSIGN, expression.

number_declaration : identifier list, COLON, constant, ASSIGN,
universal_static_expression.

3.3.1 Type Declarations

type_declaration : full_type_declaration; incomplete_type_declaration;
private_type_declaration.

full_type_declaration : TYPE, identifier, discriminant_part option, IS,
type_definition, SEMICOLON.

type_definition : enumeration_type_definition; integer_type_definition;
real_type_definition; array_type_definition;
record_type_definition; access_type_definition;
derived_type_definition.

3.3.2 Subtype Declarations

subtype_declaration : subtype symbol, identifier, IS,
subtype_indication.

subtype_indication : type_mark, constraint option.

type_mark : type_name; subtype_name.

constraint : range_constraint; floating_point_constraint;
fixed_point_constraint; index_constraint; discriminant_constraint.

3.4 Derived Types

derived_type_definition : NEW, subtype_indication.

3.5 Scalar Types

range_constraint : range symbol, range_expression.

range_expression : range_attribute; simple_expression, ddot, simple_expression.

3.5.1 Enumeration Types

enumeration_type_definition : LPAREN, enumeration_literal_specification list, RPAREN.

enumeration_literal_specification : enumeration_literal.

enumeration_literal : identifier; character_literal.

3.5.4 Integer Types

integer_type_definition : range_constraint.

3.5.6 Real Types

real_type_definition : floating_point_constraint; fixed_point_constraint.

3.5.7 Floating Point Types

floating_point_constraint : floating_accuracy_definition, range_constraint option.

floating_accuracy_definition : digits symbol, static_simple_expression.

3.5.9 Fixed Point Types

fixed_point_constraint : fixed_accuracy_definition, range_constraint option.

fixed_accuracy_definition : delta symbol, static_simple_expression.

3.6 Array Types

array_type_definition : unconstrained_array_definition; constrained_array_definition.

unconstrained_array_definition : array symbol, index_subtype_definition list, OF, component_subtype_definition.

index_subtype_definition : type_mark, range symbol, BOX.

index_constraint : LPAREN, discrete_range list, RPAREN.

discrete_range : discrete_subtype_indication; range.

3.7 Record Types

record_type_definition : RECORD, component_list, END, RECORD.

component_list : component_declaration, component_declaration repeated;
    component_declaration repeated, variant_part; null, SEMICOLON.

component_declaration : identifier list, COLON,
    component_subtype_definition, assignment option, SEMICOLON.

component_subtype_definition : subtype_indication.

3.7.1 Discriminants

discrminant_part : LPAREN, discriminant_specification sequence, RPAREN.

discrimination_specification : identifier list, COLON, type_mark,
    assignment option.

3.7.2 Discriminant Constraints

discriminant_constraint : LPAREN, discriminant_association list,
    RPAREN.

discriminant_association : discriminant_name_part option, expression.

discriminant_name_part : discriminant_simple_name, ARROW;
    discriminant_simple_name, vertical bar symbol,
    discriminant_name_part.

3.7.3 Variant Parts

variant_part : case symbol, discriminant_simple_name, IS, variant,
    variant repeated, END, case symbol.

variant : when symbol, choice, option repeated, ARROW, component_list.

option : vertical bar symbol, choice.

choice : simple_expression; discrete_range; others symbol;
    component_simple_name.

3.8 Access Types

```
access_type_definition : access symbol, subtype_indication.
```

3.8.1 Incomplete Type Declarations

```
incomplete_type_declaration : TYPE, identifier, discriminant_part
     option, SEMICOLON.
```

3.9 Declarative Parts

```
declarative_part : basic_declarative_item repeated,
     later_declarative_item repeated.

basic_declarative_item : basic_declaration; representation_clause;
     use_clause.

later_declarative_item : body; subprogram_declaration;
     package_declaration; task_declaration; generic_declaration;
     use_clause; generic_instantiation.

body : proper_body; body_stub.

proper_body : subprogram_body; package_body; task_body.
```

4.1 Names

```
name : simple_name; character_literal; operator_symbol;
     indexed_component; slice; selected_component; attribute.

simple_name : identifier.

prefix : name; function_call.
```

4.1.1 Indexed Components

```
indexed_component : prefix, LPAREN, expression sequence, RPAREN.
```

4.1.2 Slices

```
slice : prefix, LPAREN, discrete_range, RPAREN.
```

4.1.3 Selected Components

```
selected_component : prefix_selector.
```

selector : simple_name; character literal; operator symbol; all symbol.

4.1.4 Attributes

attribute : prefix, APOSTROPHE, attribute_designator.

attribute_designator : simple_name, optional attrsuffix.

attrsuffix : LPAREN, universal_static_expression, RPAREN.

4.2 Literals

4.3 Aggregates

aggregate : LPAREN, component_association list, RPAREN.

component_association : choice_part option, expression.

choice_part : choice, option repeated, ARROW.

4.4 Expressions

expression : relation, expression_suffix option.

expression_suffix : short_circuit_control_form, relation;
    logical_operator, relation.

logical_operator : AND; OR; XOR.

short_circuit_control_form : AND, THEN; OR, ELSE.

relation : simple_expression; simple_expression, relational_operator,
    simple_expression; simple_expression, NOT option, IN,
    range_expression; simple_expression, NOT option, IN, type_mark.

simple_expression : unary_adding_operator option, term, se_suffix
    option.

se_suffix : binary_adding_operator, term.

term : factor, term_suffix option.

term_suffix : multiplying_operator, factor.

factor : primary, exponential option; abs symbol, primary; NOT;
    primary.

exponential : DSTAR, primary.

primary : numeric_literal; NULL; aggregate; string_literal; name;
allocator; function_call; type_conversion; qualified_expression;
LPAREN, expression, RPAREN.

4.5 Operators and Expression Evaluation

logical_operator : AND; OR; XOR.

relational_operator : EQUAL; INEQUALITY; LT; LE; GT; GE.

binary_adding_operator : PLUS; HYPHEN; ampersand symbol.

unary_adding_operator : PLUS; HYPHEN.

multiplying_operator : STAR; SLASH; mod symbol; rem symbol.

highest_precedence_operator : DSTAR; abs symbol; NOT.

4.6 Type Conversions

type_conversion : type_mark, LPAREN, expression, RPAREN.

4.7 Qualified Expressions

qualified_expression : type_mark, APOSTROPHE, LPAREN, expression,
RPAREN; type_mark, APOSTROPHE, aggregate.

4.8 Allocators

allocator : NEW, subtype_indication; NEW, qualified_expression.

5.1 Simple and Compound Statements -- Sequences of Statements

sequence_of_statements : statement, statement repeated.

statement : label repeated, simple_statement; label repeated,
compound_statement.

simple_statement : null_statement; assignment_statement;
procedure_call_statement; exit_statement; return_statement;
goto_statement; entry_call_statement; delay_statement;
abort_statement; raise_statement; code_statement.

compund_statement : if_statement; case_statement; loop_statement;
block_statement; accept_statement; select_statement.

label : LLABEL, label_simple_name, RLABEL.

null_statement : NULL, SEMICOLON.

5.2 Assignment Statement

assignment_statement : variable_name, assignment.

5.3 If Statements

if_statement : IF, condition, THEN, sequence_of_statements, elsif_part repeated, else_part option, END, IF, SEMICOLON.

elsif_part : elsif symbol, condition, THEN, sequence_of_statements.

else_part : ELSE, sequence_of_statements.

condition : boolean_expression.

5.4 Case Statements

case_statement : case symbol, expression, IS,
case_statement_alternative, case_statement_alternative repeated,
END, case symbol.

case_statement_alternative : variant, sequence_of_statements.

5.5 Loop Statements

loop_statement : loop_prefix option, iteration_scheme option, loop symbol, sequence_of_statements, END, loop symbol, loop_simple_name option, SEMICOLON.

reverse : reverse symbol.

loop_prefix : loop_simple_name, COLON.

iteration_scheme : while symbol, condition; FOR,
loop_parameter_specification.

loop_parameter_specification : identifier, IN, reverse option,
discrete_range.

5.6 Block Statements

block_statement : block_prefix option, declare_part option, BEGIN,
sequence_of_statements, exception_part option, END,
block_simple_name option, SEMICOLON.

block_prefix : block_simple_name, COLON.

declare_part : declare symbol, declarative_part.

exception_part : exception symbol, exception_handler,
exception-handler repeated.

5.7 Exit Statements

exit_statement : exit symbol, loop_name option, when_part option,
SEMICOLON.

when_part : when symbol, condition.

5.8 Return Statements

return_statement : return symbol, expression option, SEMICOLON.

5.9 Goto Statements

goto_statement : goto symbol, label_name.

6.1 Subprogram Declarations

subprogram_declaration : subprogram_specification, SEMICOLON.

subprogram_specification : procedure symbol, identifier, formal_part
option; function symbol, designator, formal_part option, return
symbol, type_mark.

designator : identifier; operator_symbol.

operator_symbol : string_literal.

formal_part : LPAREN, parameter_specification sequence, RPAREN.

parameter_specification : identifier list, COLON, mode, type_mark,
assignment option.

mode : IN option, OUT option.

6.3 Subprogram Bodies

subprogram_body : subprogram_specification, IS, declarative_part
option, BEGIN, sequence_of_statements, exception_part, END,
designator option, SEMICOLON.

6.4 Subprogram Calls

procedure_call_statement : procedure_name, actual_parameter_part option.

function_call : function_name, actual_parameter_part option.

actual_parameter_part : LPAREN, parameter_association list, RPAREN.

parameter_association : formal_part option, actual_parameter.

formal_part : formal_parameter, ARROW.

formal_parameter : parameter_simple_name.

actual_parameter : expression; variable_name; type_mark, LPAREN, variable_name, RPAREN.

7.1 Package Structure

package_declaration : package_specification.

package_specification : package, identifier, IS, basic_declarative_item option, private_part option, END, package_simple_name option.

private_part : private symbol, basic_declarative_item repeated.

package_body : PACKAGE, BODY, package_simple_name, IS, declarative_part option, BEGIN, sequence_of_statements, exception_part option, END, package_simple_name option, SEMICOLON.

7.4 Private Type and Deferred Constant Declarations

private_type_declaration : TYPE, identifier, discriminant_part option, IS, limited option, private symbol, SEMICOLON.

deferred_constant_declaration : identifier list, COLON, constant, type_mark, SEMICOLON.

8.4 Use Clauses

use_clause : USE, package_name, package_name repeated, SEMICOLON.

8.5 Renaming Declarations

renaming_declaration : identifier, COLON, type_mark, RENAMES,
object_name, SEMICOLON; identifier, exception symbol, RENAMES,
exception_name, SEMICOLON; PACKAGE, identifier, RENAMES,
package_name, SEMICOLON; subprogram_specification, RENAMES,
subprogram_or_entry_name, SEMICOLON.

9.1 Task Specifications and Task Bodies

task_declaration : task_specification, SEMICOLON.

task_specification : TASK, TYPE option, identifier, is_part option.

is_part : IS, entry_declaration option, representation_clause option,
END, task_simple_name option.

task_body : TASK, BODY, task_simple_name, IS, declarative_part option,
BEGIN, sequence_of_statements, exception_part option, END,
task_simple_name option, SEMICOLON.

9.5 Entries, Entry Calls, and Accept Statements

entry_declaration : entry symbol, identifier, range_part option,
formal_part option, SEMICOLON.

entry_call_statement : entry_name, actual_parameter_part option,
SEMICOLON.

accept_statement : accept symbol, entry_simple_name, index_part option,
formal_part option, accept_body option.

entry_index : expression.

range_part : LPAREN, discrete_range, RPAREN.

index_part : LPAREN, entry_index, RPAREN.

accept_body : do symbol, sequence_of_statements, END,
entry_simple_name option.

9.6 Delay Statements, Duration, and Time

delay_statement : delay symbol, simple_expression, SEMICOLON.

9.7 Select Statements

select_statement : selective-wait; conditional_entry_call;
timed_entry_call.

9.7.1 Selective Waits

selective_wait : SELECT, select_alternative, or_part option, else_part option, END, SELECT, SEMICOLON.

or_part : OR, select_alternative.

select_alternative : when_part option, selective_wait_alternative.

when_part : when symbol, condition, ARROW.

selective_wait_alternative : accept_alternative, delay_alternative, terminate_alternative.

accept_alternative : accept_statement, sequence_of_statements option.

delay_alternative : delay_statement, sequence_of_statements option.

terminate_alternative : terminate symbol, SEMICOLON.

9.7.2 Conditional Entry Calls

conditional_entry_call : SELECT, entry_call_statement, optional sequence_of_statements, ELSE, sequence_of_statements, END, SELECT, SEMICOLON.

9.7.3 Timed Entry Calls

timed_entry_call : SELECT, entry_call_statement, sequence_of_statements option, OR, delay_alternative, END, SELECT, SEMICOLON.

9.10 Abort Statements

abort_statement : abort symbol, task_name list, SEMICOLON.

10.1 Compilation Units -- Library Units

compilation : compilation_unit repeated.

compilation_unit : context_clause, library_unit; context_clause, secondary_unit.

library_unit : subprogram_declaration; package_declaration;
generic_declaration; generic_instantiation; subprogram_body.

secondary_unit : library_unit_body; subunit.

library_unit_body : subprogram_body; package_body.

10.1.1 Context Clauses — With Clauses

context_clause : with_part repeated.

with_part : with_clause, use_clause repeated.

with_clause : with symbol, unit_simple_name list, SEMICOLON.

10.2 Subunits of Compilation Units

body_stub : subprogram_specification, IS, separate symbol, SEMICOLON;
PACKAGE, BODY, package_simple_name, IS, separate symbol,
SEMICOLON; TASK, BODY, task_simple_name, IS, separate symbol,
SEMICOLON.

subunit : separate symbol, LPAREN, parent_unit_name, RPAREN,
proper_body.

11.1 Exception Declarations

exception_declaration : identifier list, COLON, exception symbol.

11.2 Exception Handlers

exception_handler : when symbol, exception_choice, other_choices
option, ARROW, sequence_of_statements.

other_choices : vertical bar symbol, exception_choice.

exception_choice : exception_name; others symbol.

11.3 Raise Statements

raise_statement : raise symbol, exception_name option, SEMICOLON.

12.1 Generic Declarations

generic_declaration : generic_specification, SEMICOLON.

generic_specification : generic_formal_part, subprogram_specification;
    generic_formal_part, package_specification.

generic_formal_part : generic symbol, generic_parameter_declaration
    repeated.

generic_parameter_declaration : identifier list, COLON, generic_mode
    option, type_mark, assignment option, SEMICOLON; TYPE, identifier,
    IS, generic_type_definition, SEMICOLON; private_type_declaration;
    with symbol, subprogram_specification, sub_is_part option.

generic_mode : IN, OUT option.

sub_is_part : IS, name; IS, BOX.

generic_type_definition : LPAREN, BOX, RPAREN; range symbol, BOX;
    digits symbol, BOX; delta symbol, BOX; array_type_definition;
    access_type_definition.

12.3 Generic Instantiation

generic_instantiation : PACKAGE, identifier, IS, NEW,
    generic_package_name, generic_actual_part option, SEMICOLON;
    procedure symbol, identifier, IS, NEW, generic_procedure_name,
    generic_actual_part option; FUNCTION, designator, IS, NEW,
    generic_function_name, generic_actual_part option, SEMICOLON.

generic_actual_part : LPAREN, generic_association list, RPAREN;

generic_association : generic_formal_part option,
    generic_actual_parameter.

generic_formal_part : generic_formal_parameter, ARROW.

generic_formal_parameter : parameter_simple_name; operator_symbol.

generic_actual_parameter : expression; variable_name; subprogram_name;
    entry_name; type_mark.

13.1 Representation Clauses

representation_clause : type_representation_clause; address_clause.

type_representation_clause : length_clause;
    enumeration_representation_clause; record_representation_clause.

13.3 Enumeration Representation Clauses

enumeration_representation_clause : FOR, type_simple_name, USE,
aggregate, SEMICOLON.

13.4 Record Representation Clauses

record_representation_clause : FOR, type_simple_name, USE, RECORD,
alignment_clause option, component_clause option, END, RECORD,
SEMICOLON.

alignment_clause : AT, mod symbol, static_simple_expression,
SEMICOLON.

component_clause : component_name, AT, static_simple_expression,
range symbol, static_range, SEMICOLON.

13.5 Address Clause

address_clause : FOR, simple_name, USE, AT, simple_expression,
SEMICOLON.

13.8 Machine Code Insertions

code_statement : type_mark, APOSTROPHE, record_aggregate, SEMICOLON.

Appendix B: W-grammar B

Metaproductions

(B.A) CHAR :: graphic_character.

(B.B) LETTER :: a; b; c; d; e; f; g; h; i; j; k; l; m; n; o; p; q; r;
s; t; u; v; w; x; y; z.

(B.C) DIGIT :: 1; 2; 3; 4; 5; 6; 7; 8; 9; 0.

(B.D) NUMBER :: zero; one; two; three; four; five; six; seven; eight;
nine.

(B.E) NOTION :: CHAR; NOTION CHAR.

(B.F) EMPTY :: .

(B.G) NOTETY :: NOTION; EMPTY.

(B.H) COLLATING SEQUENCE :: 01234567890abcdefghijklmnopqrstuvwxyz_.

(B.I) IDENTIFIER :: LETTER; IDENTIFIER underscore; IDENTIFIER LETTER;
IDENTIFIER DIGIT.

(B.J) NAME :: name LETTER; NAME name CHAR.

(B.K) LIBRARY ::
nextentry library entry endlibrary;
nextentry library entry LIBRARY.

(B.L) ID :: library entry.

(B.M) IDS :: id ID; IDS id ID.

(B.N) IDETY :: ID; EMPTY.

(B.O) FORMAL PART :: parameter specification group.

(B.P) TYPE MARK :: type designator.

(B.Q) PSEQUENCE :: parameter specification sequence.

(B.R) DESCRIPTION :: kindof IDENTIFIER1 parameter NOTETY return
IDENTIFIER2 CONTENTS.

(B.S) CONTENTS :: contains ID repeated.

(B.T) UNDERSCORE :: _.

Hyper-rules

(B.1) graphic_character : basic graphic_character; lower_case symbol;
other_special_character symbol.

(B.2) basic_graphic_character : upper_case symbol; DIGIT symbol;
special_character symbol; space_character.

(B.3) basic_character : basic_graphic_character; format_effector;

(B.4) upper_case : capital LETTER.

(B.5) special_character : quotation; sharp; ampersand; apostrophe;
left parenthesis; right parenthesis; star; plus; comma;
hyphen; dot; slash; colon; semicolon; less than; equal;
greater than; underline; vertical bar.

(B.6) lower_case : letter LETTER.

(B.7) other_special_character : exclamation mark; dollar; percent;
question mark; commercial at; left square bracket; back-slash;
right square bracket; circumflex; grave accent; left brace;
right brace; tilde.

(B.8) NOTION option : NOTION; EMPTY.

(B.9) NOTION repeated : NOTION repeated, NOTION; NOTION; EMPTY.

(B.10) NOTION list : NOTION; NOTION list, comma symbol, NOTION.

(B.11) NOTION sequence : NOTION;
NOTION sequence, semicolon symbol, NOTION.

(B.12) NOTION pack : left parenthesis symbol, NOTION list,
right parenthesis symbol.

(B.13) NOTION group : left parenthesis symbol, NOTION sequence,
right parenthesis symbol.

(B.14) library entry : NAME DESCRIPTION.

(B.15) letter_or_digit : letter LETTER symbol; NUMBER symbol.

(B.16) NOTETY1 print :
where NOTETY1 is EMPTY;
NOTETY2 print, underscore symbol,
where NOTETY1 is UNDERSCORE NOTETY2;
NOTETY2 print, letter LETTER symbol,
where NOTETY1 is LETTER NOTETY2;
NOTETY2 print, DIGIT symbol,
where NOTETY1 is DIGIT NOTETY2.

```
(B.16) true : EMPTY.

(B.17) where NOTETY is NOTETY : true.

(B.18) where NOTION is in NOTETY1 NOTION NOTETY2 : true.

(B.19) where NOTION is not in EMPTY : true.

(B.20) where EMPTY is not in NOTION : true.

(B.21) where NOTETY1 or NOTETY2 : where NOTETY1; where NOTETY2.

(B.22) where NOTETY1 and NOTETY2 : where NOTETY1, where NOTETY2.

(B.23) where NOTION1 is not in NOTION2 :
          where NOTETY1 CHAR1 is NOTION1 and NOTETY2 CHAR2 is NOTION2,
             where CHAR1 is not CHAR2 and NOTION1 is not in NOTETY2;
          where NOTETY1 CHAR1 is NOTION1 and NOTETY2 CHAR2 is NOTION2,
             where NOTETY1 is not in NOTETY2.

(B.24) where NOTION1 and NOTION2 : where NOTION1, where NOTION2.

(B.25) where CHAR1 is not CHAR2 :
          where CHAR1 precedes CHAR2 in COLLATING SEQUENCE;
          where CHAR2 precedes CHAR1 in COLLATING SEQUENCE.

(B.26) where CHAR1 precedes CHAR2 in NOTION :
          where NOTETY1 CHAR1 NOTETY2 CHAR2 NOTETY3 is NOTION.

(B.27) where NOTION1 is not NOTION2 :
          where NOTETY1 CHAR1 is NOTION1 and NOTETY2 CHAR2 is NOTION2
             and NOTETY1 is not NOTETY2;
          where NOTETY1 CHAR1 is NOTION1 and NOTETY2 CHAR2 is NOTION2
             and CHAR1 is not CHAR2.

(B.28) where NOTION1 describes NOTION2 :
          where NOTION1 is NOTION3 NOTETY1,
          where NOTION3 is name ALPHA1 NOTION4,
          where NOTION2 is ALPHA1 NOTETY2,
          where NOTION4 describes NOTETY3.

(B.29) where NOTION1 describes EMPTY :
          where NOTION1 is kindof NOTION2.

(B.30) where NOTION1 kind NOTION2 :
          where NAME kindof NOTION2 parameter NOTETY is NOTION1.

(B.31) where NOTION1 parameter NOTION2 :
          where NAME kindof NOTETY1 parameter NOTION2 return NOTETY2 is
             NOTION1.
```

(B.32) where NOTION1 return NOTION2 :
where NAME kindof NOTETY1 parameter NOTETY2 return NOTION2
contains NOTETY3 is NOTION1.

(B.33) where NOTION1 contains NOTION2 :
where NAME kindof NOTETY1 parameter NOTETY2 return NOTION3 is
NOTION1 and NOTETY3 contains NOTION2 NOTETY4 is NOTION3,
where EMPTY is NOTETY4 or contains NOTETY5 is NOTETY4.

(B.34) LIBRARY compilation : LIBRARY compilation unit repeated.

(B.35) LIBRARY compilation unit :
LIBRARY context clause, LIBRARY library unit ID;
LIBRARY context clause, LIBRARY secondary unit ID.

(B.36) LIBRARY library unit ID :
LIBRARY subprogram declaration ID;
LIBRARY package declaration ID;
LIBRARY generic declaration ID;
LIBRARY generic instantiation ID;
LIBRARY subprogram body ID, where ID is not in LIBRARY.

(B.37) LIBRARY secondary unit ID :
LIBRARY library unit body ID, where ID is in LIBRARY;
LIBRARY subunit ID, where ID is in LIBRARY.

(B.38) library unit body ID : subprogram body ID; package body ID.

(B.39) LIBRARY context clause :
LIBRARY IDS with clause LIBRARY IDS use part repeated.

(B.40) LIBRARY IDS use part :
use symbol, LIBRARY IDS use clause repeated.

(B.41) LIBRARY IDS with clause :
with symbol, LIBRARY IDS with part, SEMICOLON.

(B.42) LIBRARY IDS with part : IDENTIFIER,
where ID describes IDENTIFIER,
where NOTETY1 nextentry ID NOTETY2 is LIBRARY,
where NOTETY3 id ID NOTETY4 is IDS,
LIBRARY NOTETY3 NOTETY4 with part.

(B.43) LIBRARY with part : EMPTY.

(B.44) LIBRARY IDS use clause :
use symbol, LIBRARY IDS package list, SEMICOLON.

```
(B.45) LIBRARY IDS package : IDENTIFIER,
          where ID kind package,
          where ID describes IDENTIFIER,
          where NOTETY1 nextentry ID NOTETY2 is LIBRARY,
          where NOTETY3 id ID NOTETY4 is IDS.

(B.46) LIBRARY subprogram declaration ID : subprogram specification ID,
          where NOTETY1 nextentry ID NOTETY2 is LIBRARY,
          where NOTETY3 id ID NOTETY4 is IDS,
          LIBRARY NOTETY3 NOTETY4 with part.

(B.47) subprogram specification ID :
          procedure symbol, IDENTIFIER, FORMAL PART option,
             where ID describes IDENTIFIER,
             where ID kind procedure,
             where ID parameter FORMAL PART;
          function symbol, IDENTIFIER, FORMAL PART option,
             return symbol, TYPE MARK,
             where ID describes IDENTIFIER,
             where ID kind function,
             where ID parameter FORMAL PART,
             where ID return TYPE MARK.

(B.48) LIBRARY subprogram specification :
          procedure symbol, IDENTIFIER, FORMAL PART option,
             LIBRARY validate FORMAL PART;
          function symbol, IDENTIFIER, FORMAL PART option,
             return symbol, TYPE MARK,
             where ID describes TYPE MARK,
             where ID kind type,
             where ID is in LIBRARY.

(B.49) LIBRARY validate FORMAL PART :
          where ( PSEQUENCE ) is FORMAL PART,
          LIBRARY validate PSEQUENCE.

(B.50) LIBRARY validate PSEQUENCE1 :
          where PSEQUENCE2 semicolon PSEQUENCE3 is PSEQUENCE1,
             where PSEQUENCE3 is parameter specification,
             LIBRARY validate PSEQUENCE2,
             LIBRARY validate PSEQUENCE3;
          where PSEQUENCE is parameter specification,
             where IDENTIFIER list colon mode TYPE MARK NOTETY1 is
                PSEQUENCE,
             where ID describes TYPE MARK,
             where NOTETY2 nextentry ID NOTETY3 is LIBRARY,
             where ID kind type,
             where NOTETY1 is TYPE MARK assignment.

(B.51) parameter specification :
          IDENTIFIER list, colon symbol, mode, TYPE MARK,
             TYPE MARK assignment option.
```

(B.52) mode : in symbol option, out symbol option.

(B.53) LIBRARY1 subunit ID :
separate symbol, left paren symbol, IDENTIFIER,
right paren symbol, LIBRARY2 proper body ID2,
where ID contains ID2,
where ID describes IDENTIFIER,
where LIBRARY2 ID NOTETY is LIBRARY1.

(B.54) proper body : subprogram body; package body; task body.

Bibliography

1. Bedford Computer Center. BCC UNIX New User's Guide. MITRE Corporation, Bedford MA, 27 August 1984.

2. Booth, K. H. V. "Meta Language," Encyclopedia of Computer Science, edited by Anthony Ralston. New York: Van Nostrand Reinhold, 1976.

3. Cameron, Robert D. and M. Robert Ito. "Grammar-Based Definition of Metaprogramming Systems," ACM Transactions on Programming and Systems, 6: 20-54 (January 1984).

4. Cleaveland, J. Craig and Robert C. Uzgalis. Grammars for Programming Languages. New York: Elsevier North-Holland, 1977.

5. Department of Defense. Ada Programming Language. ANSI/MIL-STD 1815A. Philadelphia: Naval Publications and Forms Center, 22 January 1983.

6. Fleck, A. C. and R. S. Limaye. "Formal Semantics and Abstract Properties of String Pattern Operations and Extended Formal Language Description Mechanisms," SIAM Journal on Computing, 12: 166-168 (February 1983).

7. Hopcroft, John E. and Jeffrey D. Ullman. Introduction to Automata Theory, Languages, and Computation. Reading MA: Addison-Wesley Publishing Company, 1979.

8. McCracken, D. D. "Backus-Naur Form," Encyclopedia of Computer Science, edited by Anthony Ralston. New York: Van Nostrand Reinhold, 1976.

9. Reps, Thomas et al. "Incremental Context-Dependent Analysis for Language-Based Editors," ACM Transactions on Programming Languages and Systems, 5: 449-477 (July 1983).

10. Sethi, Ravi. "Control Flow Aspects of Semantics-Directed Compiling," ACM Transactions on Programming Languages and Systems, 5: 554-595 (October 1983).

11. Tanaka, Eiichi and King-Sun Fu. "Correction to Error-Correcting Parsers for Formal Languages," IEEE Transactions on Computers, C-31: 327-328 (April 1972).

12. ———. "Error-Correcting Parsers for Formal Languages," IEEE Transactions on Computers, C-27: 605-616 (July 1978).

13. Tennent, R. D. "The Denotational Semantics of Programming Languages," Communications of the ACM, 19: 437-453 (August 1976).

14. Uhl, Jurgen et al. An Attribute Grammar for the Semantic Analysis of Ada. Berlin: Springer-Verlag, 1982.

15. Wegner, P. "Vienna Definition Language," Encyclopedia of Computer Science, edited by Anthony Ralston. New York: Van Nostrand Reinhold, 1976.

16. Woffinden, Maj Duard S. Department of Electrical Engineering and Computer Science. Personal Interview. Air Force Institute of Technology, Wright-Patterson AFB OH, 8 February 1986.

VITA

First Lieutenant Roy A. Flowers was born on 6 July 1957 at Johnson AFB, Japan. He graduated from Waite High School in Toledo, Ohio, in 1975 and attended Mount Vernon Nazarene College until May 1978 when he enlisted in the USAF. He completed basic training and worked as a computer programmer at the Air Force Manpower and Personnel Center, Randolph AFB, TX until August 1980. In February 1980 he was selected to participate in the Airman Education and Commissioning Program at the Ohio State University from which he received the degree of Bachelor of Science in Electrical Engineering in December 1982. He completed OTS and received a commission in April 1983. He then served as a computer engineer at Electronic Systems Division, Hanscom AFB, MA until entering the School of Engineering, Air Force Institute of Technology, in June 1985. He married the former Nancy Kean in January 1977 and they have three children: Royanna, Ryan, and Heather.

Permanent address: 178 Fornoff Road
Columbus, Ohio 43207

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

AD-A177802

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): AFIT/GCE/ENG/86D-9	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: School of Engineering; 6b. OFFICE SYMBOL (If applicable): AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code): Air Force Institute of Technology, Wright-Patterson AFB, Ohio 45433	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
A W-GRAMMAR FOR ADA

PERSONAL AUTHOR(S)
Roy A. Flowers, B.S.E.E., 1st Lt, USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
MS Thesis	FROM _____ TO _____	1986 December	78

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Ada, Grammars, Semantics, Syntax, Context Sensitive Grammars, Phrase Structure Grammars
09	02		

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

Approved for public release: IAW AFR 190-1.
LYNN E. WOLAVER
Dean for Research and Professional Development
Air Force Institute of Technology (AFIT)
Wright-Patterson AFB OH 45433

Thesis Chairman: James W. Howatt, Captain, USAF
Assistant Professor of Computer Systems

This thesis explores the formal definition of the syntax and static semantics of the Ada programming language. Several notational forms were compared and the particular notational form chosen is a double level grammar called the the W-grammar. W-grammars were first used in the formal definition of Algol 68. Two W-grammars are presented. The first W-grammar is a translation of the modified BNF notation used in the Ada Language Reference Manual, and the second demonstrates the description of Ada's static semantics in W-grammar format.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: James W. Howatt, Captain, USAF	22b. TELEPHONE (Include Area Code): 513-255-6913 / 22c. OFFICE SYMBOL: AFIT/ENG

DD Form 1473, JUN 86 — Previous editions are obsolete. — SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

END

4-87

DTIC