TRANSLATOR WRITER SYSTEMS

by

Stuart A. Odom

Project Submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science and Applications

APPROVED:

Dr. Thomas F. Reid, Chairman

Dr. William D. May

Richard E. Schneider

Date: June 1992
Blacksburg, Virginia

TRANSLATOR WRITER SYSTEMS

by

Stuart A. Odom

Committe Chairman: Thomas F. Reid
Computer Science

(ABSTRACT)

The design, structure, and use of a Translator Writer System (TWS) is analyzed. The major data structures are documented, with a view toward their use in the generation of a recursive descent parser. Module interaction within the TWS, and applications derived from it, are clearly defined. Sufficient background is provided on attributed grammars to make use of the TWS in a practical application.

The TWS is redesigned to separate areas of concern within both the modules and the data structures. The modules are redefined so that each one has a specific set of tasks to perform, each one dealing with a small facet of the TWS.

ACKNOWLEDGEMENTS

The author would like to acknowledge the help of Thomas F. Reid, William D. May, and Richard E. Schneider in the preparation of this report, and for their continued dedication as educators. The knowledge gained from these faculty members has greatly enhanced his professional abilities as a Computer Scientist. The author would also like to acknowledge the faithfulness and commitment of his family throughout his post-graduate career and the preparation of this report.

TABLE OF CONTENTS

SECTION	PAGE
1. Introduction	1
1.1 Scope of Report	1
1.2 Translators in General	2
2. Translation	4
2.1 Components of a Translator	5
2.1.1 Scanner	7
2.1.2 Parser	7
2.1.3 Semantic Analyzer	10
2.1.4 Output Generator	11
2.2 Summary of the Translation Process	11
3. TWS System Overview	13
3.1 Translator Writer Systems	13
3.2 Language Syntax	15
3.3 Backus Naur Form	17
3.3.1 Generating Members of a Language	18
3.4 Extended Backus Naur Form	19
3.4.1 EBNF in Relation to the Parsing Process	20
3.5 Example Grammar	21
3.6 Attributed Grammar and Semantics	26
3.6.1 Attributes	29
3.6.2 Semantic Actions	35
3.7 A Simple Translator	41
4. TWS System Architecture	44
4.1 TWS Operation	44
4.2 TWS Modules	44
4.3 Language Specific Modules	47
4.3.1 Module TWESTree	47
4.3.2 Module TWSymbol	48
4.3.3 Module TWSymTab	49
4.3.4 Module TWUtil	50
4.3.5 Module TWSets	50
4.3.6 Module TWCodGen	51
4.3.7 Module TWTypDef	51
4.3.8 Module TWScan	55

TABLE OF CONTENTS (continued)

SECTION	PAGE
4.3.9 Module TWInOut	55
4.3.10 Module TWSupprt	55
4.4 System Flow	56
4.4.1 Main Driver	56
4.4.2 Parser	57
5. TWS Data Structures	59
5.1 Extended Syntax Tree	59
5.1.1 Definition	59
5.1.2 Implementation	62
5.1.3 EST Record Types	65
5.1.4 EST Record Definitions	67
5.2 Use of the Extended Syntax Tree	71
5.3 Symbol Table	76
5.3.1 Symbol Table and Implementation	76
5.3.2 Use of the Symbol Table	78
6. TWS Example	82
6.1 Implementing a New Language	82
6.2 Hand Calculator Overview	86
6.3 Hand Calculator EBNF	90
6.4 Hand Calculator Generation	93
6.5 Summary	94
7. Discussion	95
7.1 Analysis of the Extended Syntax Tree	95
7.1.1 Object-oriented Approach	95
7.1.2 Hierarchical Approach	96
7.1.3 Approach Taken	97
7.2 Module Analysis	99
7.3 Lessons Learned	99
7.3.1 Technical Knowledge Gained	100
7.4 Conclusion	100
REFERENCES	102
VITA	103

LIST OF FIGURES

		PAGE
Figure 2.1	Translation Process	6
Figure 2.2	Derivation Tree for (a + a) * a	9
Figure 3.1	Relationship Between TWS and Target Translator	14
Figure 3.2	Hierarchical Structure of Computer Language	16
Figure 3.3	Derivation Tree	23
Figure 3.4	Parser for Simple Grammar	24
Figure 3.5	Annotated Derivation Tree	32
Figure 3.6	Attributed Parser for Simple Grammar	33
Figure 3.7	Simple Grammar with Semantic Actions	36
Figure 3.8	Parser for Simple Grammar with Semantic Actions	38
Figure 3.9	Modules for Simple Interpreter	42
Figure 4.1	TWS Module Interdependencies	46
Figure 4.2	TWS Definition Module TWTypDef.DEF	53
Figure 4.3	TWS General System Flow	58
Figure 5.1	Simple Grammar	60
Figure 5.2	Extended Syntax Tree for <Expr>	61
Figure 5.3	EST for Concatenation	63
Figure 5.4	TWS Implementation of Concatenation	63
Figure 5.5	TWS EST for <Expr>	64
Figure 5.6	TWS EST for <Expr> shown as Data Structures	66
Figure 5.7	Generated Parser for Simple Grammar	74
Figure 5.8	Definition Module G0Decl.DEF	79
Figure 5.9	Implementation Module G0Decl.MOD	81
Figure 6.1	Generic TWS-Based Application Flow	85
Figure 6.2	Hand Calculator Module Interaction	88
Figure 6.3	EBNF Description of Hand Calculator	91

LIST OF FIGURES (continued)

PAGE

Figure 7.1 Current TWS Tree Implementation 98

Figure 7.2 Original TWS Tree Implementation 98

LIST OF TABLES

	PAGE
Table 3.1 Common TWS Directives	27
Table 3.2 EBNF Input File Format	28
Table 3.3 Module Functions for Simple Interpreter	43
Table 5.1 ESTNode Record Fields	68
Table 5.2 SymbolNode Record Fields	69
Table 5.3 AttributeNode Record Fields	70
Table 6.1 Hand Calculator Module Descriptions	89

1. Introduction

In the Fall 1989 semester at Virginia Tech's Northern Virginia facility, the author took the Translator Design and Construction course, taught by Dr. Thomas F. Reid. The most beneficial aspect of the course was the semester-long project of modifying and using a Translator Writer System (TWS), developed by Dr. Reid. The TWS is a Modula-2 program which accepts a description of a language in an attributed grammar based on Extended Backus Naur Form (EBNF), and produces as output the source code for a Modula-2 module with imbedded semantics for a parser for that language, as well as the initialized data structures to automate its scanner. The input to the TWS is an experiment in developing a general purpose language for translators. The major benefit received from this project was a greater understanding how to provide automated support for the translation process.

This project and report extends the work on the TWS by thoroughly understanding the concepts of abstract syntax trees (AST's) for translator systems. In addition, this report serves as a user's guide and programmer's manual.

1.1 Scope of Report

This report is written with the assumption that the reader is familiar with programming language theory, as well as the parsing and compilation processes. The reader is directed to [BARR79] or [AHO77] for a comprehensive textbook on translators. The parsing process is explained only to the level that it will assist in understanding the Translator Writer System. For further detail on compilers and the parsing process, it is suggested that the sources cited in the REFERENCES section be consulted.

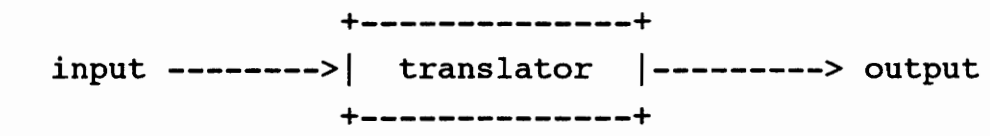
This report is a detailed explanation of the Translator Writer System with

emphasis on building and using its abstract syntax tree. There is also a detailed description of its other major data structure, the symbol table. Additionally, each module of the TWS is explained so the reader gains a full understanding of their function and interaction.

In addition to the technical description of the TWS, this report also explains to the novice how to use the TWS. The input is explained, and examples are shown of what output may be expected by the user. There is also an example toward the end of this report of how a person with a programming background can incorporate the parser generated by the TWS into a larger piece of software, in this case, a simple interpreter.

1.2 Translators in General

A translator accepts as input a potential member of an input language and produces a member of an output language, if the input is valid.



In theory, the input and output languages can be anything. However, if they are formally defined, then translators can be automatically built and verified.

Translators are almost as old as computer science. Fundamental is the concept of executing a stored program. Initially, the stored programs were written directly into the machine's hardware by manipulating toggle switches. The first abstraction was giving names to portions of instructions (such as opcode, address, etc.) giving rise to assembly languages. Other abstractions and languages followed (addresses - assemblers; expressions and statements - FORTRAN and COBOL; data - Pascal; procedures - Ada

and Modula-2). These languages were written in text which then had to be converted to a form which the computer could understand.

These converters are generically called translators. Translators are given special names when the input and output languages are specific types. Examples include assemblers when the input language is assembly language and the output language is machine language, and compilers when the input language is a high order language (HOL) and the output language is either assembly or machine language [REID90].

2. Translation

A translator accepts a member of an input language, and produces a member of an output language. Two examples of this are an assembler, in which the input is assembly code and the output language is binary, or machine language, and a compiler, in which the input is a program in a high-level language, such as C, and the output is either assembly or machine code. Of course, in such a case, if the output language is assembly language, then the resulting assembly code must then be translated into machine language if the program is to run.

The input language is the source and the output language is the target. Language theory looks on both as strings. In practice, either language can be a well-defined structure, such as a physical data base management system, or a bit-mapped graphical display.

This section discusses translators in general, particularly covering those tasks by which a **computer** language translator performs its task. It is in this section that we move away from the basic theory presented in the previous section, and begin to describe the practical applications of language translation. Specifically, the major components of a computer language translator are defined. It is important to keep in mind that due to the differences in input and output languages, not all translators are the same. The attributes of the input and output languages dictate how best to implement the components of a translator. This section provides the reader with a general overview with which the process of translation is performed.

2.1 Components of a Translator

In brief, the major components of a computer language translator are the *scanner*, the *parser*, the *semantic analyzer*, and the *output generator*, or *synthesizer*. The scanner converts each input string from the input language into a series of tokens. The parser generates parser trees, or their equivalent, from these tokens. It does this through applying the production rules defined in that language's grammar to determine if the input string is syntactically correct. The semantic analyzer determines whether or not the input string makes sense in the particular language. Finally, the synthesizer actually produces the output. The components are discussed in detail here. While the components may be thought of as separate stages, in practice they may all be executing in parallel and co-mingled. Figure 2.1 depicts this process of computer language translation, showing the relationship between the major modules and the major data structures. It is also an accurate depiction of the major components of the TWS.

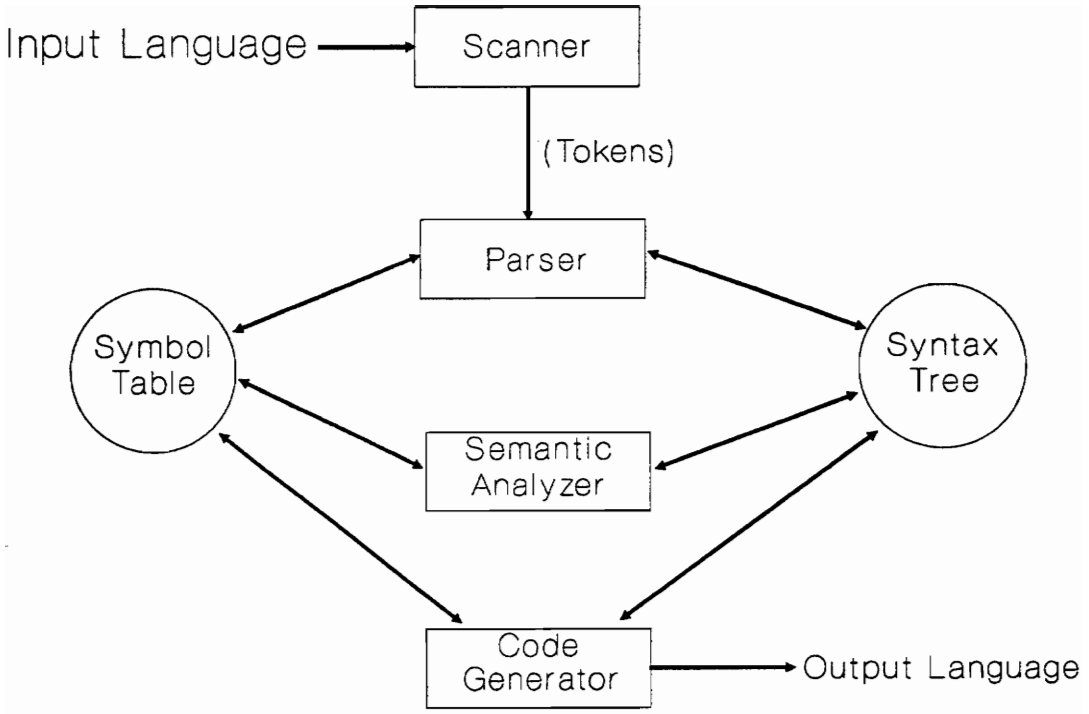


Figure 2.1 Translation Process

2.1.1 Scanner

The scanner (or lexical analyzer) transforms the input string from characters to the words or tokens of the language. The parser and scanner form a master/slave relationship. The parser asks for a token and the scanner responds by scanning the input string for the next token. The tokens fall into three categories: keywords, special symbols, and multi-valued such as integers, reals and identifiers.

In addition, the scanner generally handles the listing, throws away comments, and fields translator directives. A lower lever module, ??GetChr (the first two characters of the module names for a translator indicate which translator it is - TW for the translator writer system, CA for the hand calculator, etc.) is asked by the scanner for a character at a time and performs this operation [REID90]. The variance between scanners and text-based languages is small. ??GetChr may have to handle either static or interactive input files and the semantics of multi-valued tokens must be written.

2.1.2 Parser

As previously stated, the parser is simply an automaton which, given a string, applies production rules in the specified grammar for the language, and determines if the given string can be generated from the start symbol. Depending on the class of grammar, this automaton may be either a finite-state automaton, a finite-state push-down automaton, or a Turing machine. In the simplest terms, the parser asks the question, "Is this string a member of the language?". In other words, is this a valid Pascal (C, Modula-2, FORTRAN, etc.) program? Is it syntactically correct? If the string given is derivable from the start symbol of the grammar, then it is a syntactically correct program in the language. The parser requires support routines such as the symbol table manager. The

symbol table is a data structure which keeps track of which symbols appear in the language, usually in the form of a lexicographical list of the symbols. The parser also makes use of the syntax tree.

In order to more fully understand the parsing process, it is necessary to provide a brief introduction to *parse trees*, also called *syntax trees* or *derivation trees*. A derivation tree is an abstract data structure which conceptually displays the derivation of a string within a language. Consider the following example grammar.

```
E ---> T { "+" T } .
T ---> F { "*" F } .
F ---> "(" E ")" | "a" .
```

A derivation tree for the string (a + a) * a is presented in Figure 2.2. Once the tree is generated, the original string is obtained by doing a preorder traversal of the tree.

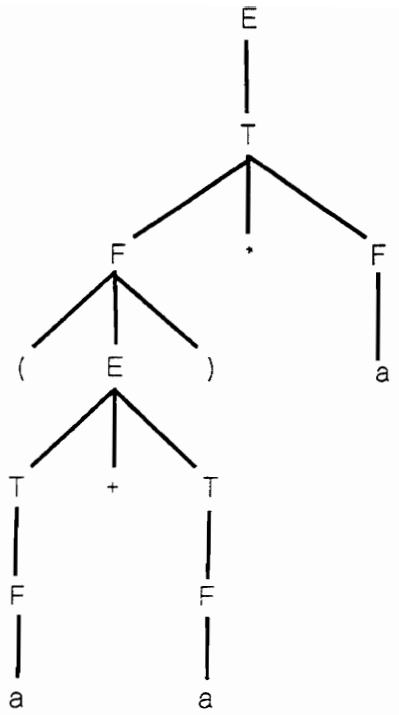


Figure 2.2 Derivation Tree for (a + a) * a

There are two types of parsers. The first type, *bottom-up parsers*, builds the tree from the leaves up. It begins with the input string (the leaves of the derivation tree) of terminals, and determines which productions should be invoked to replace a right-hand side by a left-hand side, thus building the tree from the leaves up. The following from [BARR79] best states the problem: "Given a partially constructed set of derivation trees for a right-most bottom-up parse, determine the production the right member of which fits the left-most set of roots of the trees, and that adds one more production to the derivation tree set".

The second type of parser, the *top-down parser*, builds the tree from the start symbol down. It begins with the start symbol of the grammar at the root of the derivation tree, and constructs the tree from the top down, applying productions (and backtracking, if necessary) until the entire derivation tree is complete. The leaf nodes, when traversed preorder, produce the original input string. The most popular implementation of top-down parsing is *recursive descent parsing.* In recursive descent parsing, one procedure is written for each production. The sequence of statements in the procedure mimics exactly the sequence of symbols and meta-symbols (these terms will become clear when BNF notation is discussed) in the production, so the parser always knows what to expect at each point in the parsing process. Based on what token it expects to receive from the scanner next, if it receives it, it knows exactly which production to apply next.

2.1.3 Semantic Analyzer

The semantics of an input statement refer to the meaning of the statement - does it make sense, according to the execution rules of the language? Even if a string is syntactically correct, it may not be semantically correct. For example, assigning an integer value to a character is syntactically correct for both Pascal and C, but violates type

constraints in Pascal, whereas in C it would be perfectly valid. The semantic analysis phase is usually co-mingled with the parsing phase, but may be a separate pass on the parse tree. The translator's requirements drive the form of structures that are needed for the semantic analysis. Usually, a symbol table keeps track of how a name is defined and used. The parser builds and uses structures such as parse trees or abstract syntax trees to maintain a syntactic history of the parse. It is very natural to "decorate" these trees to maintain the semantic history. A tree is said to be *augmented* if semantics or synthesis information is kept in it.

2.1.4 Output Generator

As the name implies, the output generator, or synthesizer, actually produces the target code, or output language. This phase is often called the code generation phase. Some translators produce the output in-line with the parser. This method is the case with on-the-fly interpreters, like command shells or the original BASIC. Compilers, however, produce machine code for the target machine, which is then executed independently of the compiler. The language's specifications will dictate whether a complete augmented parse tree or abstract syntax tree must be built for multiple passes or whether a single pass translator can discard parts of the tree when they are no longer needed. Optimization of the output code may also be a part of the code generation phase.

2.2 Summary of the Translation Process

The translation process has three phases. First, convert the characters in the input string to tokens. Second, build the symbol table and syntax trees from these tokens,

attaching semantic actions in the process. These first two phases are co-mingled in that when a token is retrieved, it is immediately put into the symbol table and syntax tree. Third, generate the output language from the syntax tree and symbol table. Error handling is a part of each phase.

3. TWS System Overview

This section will provide a description of how the TWS operates, within the context of language translation in general. Through a simple grammar example, the basic functionality of the TWS will be demonstrated. The necessary concepts, such as attributed grammars and semantic actions, are also presented here with an eye as to where they fit into the resulting parser.

3.1 Translator Writer Systems

A translator writer system is, itself, a translator. It produces translators. In fact, a figure of merit for a TWS is to produce itself. A TWS's source is the specification of a language (e.g. an attributed grammar for Pascal). The output then of a TWS is a translator for that language (e.g. the source code for a Pascal compiler). The relationship between a TWS and its target translator is shown in Figure 3.1. It is important to keep this view in mind as will help avoid any attempts to think of the TWS as a specific translator, or a specific parser generator.

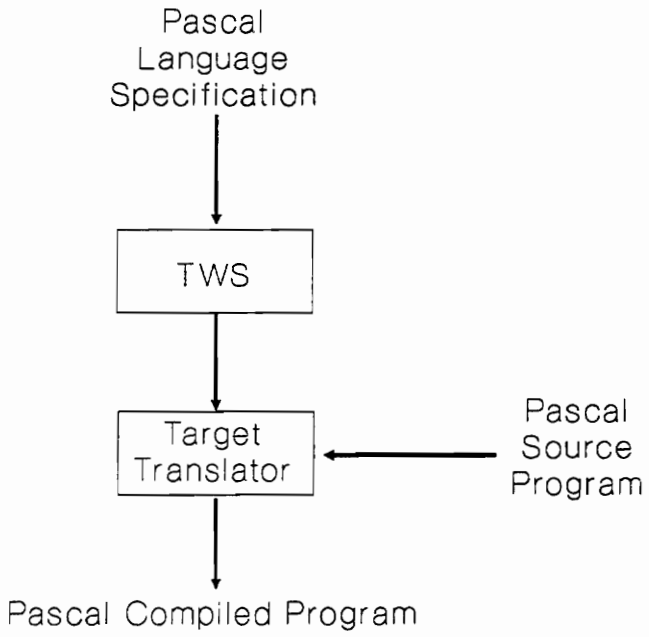


Figure 3.1 Relationship Between TWS and Target Translator

3.2 Language Syntax

In order to fully understand how the TWS functions, it is necessary to provide a brief discussion of computer languages in theory. This is because the input to the TWS is a file containing a description of a language's grammar in Extended Backus Naur Form (EBNF) notation. The concepts behind EBNF must be understood if the TWS is to be understood.

A language can be thought of as a set of sentences. A sentence is made up of a string of terminal symbols from an alphabet. These elements generate a legal sentence in the language according to the language's grammar. A parser determines whether a string satisfies that grammar. A grammar consists of a set of production rules of the form

x ---> y

where x is a nonterminal symbol, and y is a concatenation of terminal and nonterminal symbols. One nonterminal, called the start symbol, denotes which production rule to apply first in the derivation of a string. The following is an example of a production rule for a Modula-2 WHILE statement:

WhileStmt ---> WHILE Expr DO StmtList END ; .

In short, a computer language consists of syntax and semantics. The syntax of a language is defined by a formal language specified as a grammar. A grammar consists of an alphabet, a start symbol, and a set of production rules. The alphabet is made up of strings, terminal symbols, and nonterminal symbols. Figure 3.2 depicts this hierarchical view of a computer language.

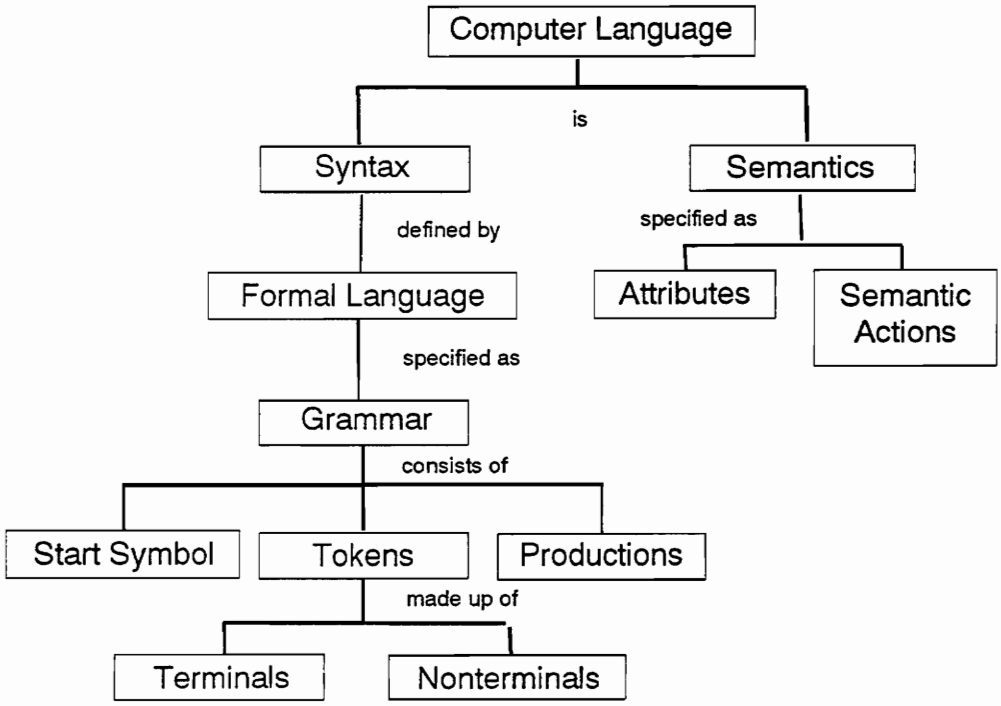


Figure 3.2 Hierarchical Structure of Computer Language

3.3 Backus Naur Form

It has been stated that a translator accepts as input a member of an input language, and generates a member of an output language. In order to do this, the translator must parse the input to be sure it is indeed a member of the input language. A grammar is associated with the input language. Part of this grammar is a set of production rules which defined which strings can be derived from which nonterminals. It is these rules that the parser uses to determine if the input string is in the input language. The parser attempts to derive the input string beginning with the start symbol for the grammar using the rules in the production set P. In order to accomplish this task, there must be some way of describing the input language using some generally accepted convention. Backus Naur Form (BNF) is the most common of these notations.

BNF is used to generate the class of context-free languages which is powerful enough to describe most programming languages. The following is an example of BNF for the declaration section of a simple programming language:

<Declarations> ::= <Declaration> | <Declarations> <Declaration> .
<Declaration> ::= "VAR" <Name-Sequence> ":" <Type> ";".
<Name-Sequence> ::= <Name> | <Name-Sequence> "," <Name> .
<Type> ::= ("INTEGER" | "REAL") .

After the notation is explained, BNF is easy to read. For example, the top line means: "<Declarations> is defined as <Declaration> or <Declarations> followed by <Declaration>". The second line means: <Declaration> is defined as the keyword "VAR" followed by a <Name-Sequence> followed by ":" followed by a <Type> followed by a ";".

There are two types of symbols: target-symbols and meta-symbols. Target

symbols are the elements of the language being defined and are enclosed in < and > or " characters. Thus, the target symbols of the above language are: <Declarations>, <Declaration>, "VAR", <Name-Sequence>, ":", <Type>, ";", <Name-Sequence>, <Name>, ",", "INTEGER", and "REAL".

The target-symbols enclosed in quotes are called terminals and will appear in the target language without the quotes. The target-symbols enclosed in < and > are called nonterminals and must appear on at least one LHS (to be defined) or will have a class of terminals supplied by the scanner (e.g., <Name>). By convention, the first LHS nonterminal is called the start symbol of the language and all members of the language are derived by starting with this start symbol.

Meta-symbols of BNF are ::=, ., |, (, and). The symbol ::= means "is defined as" and "<a> ::= b" reads "<a> is defined as b". The symbol <a> preceding the ::= is called the left-hand side (LHS) and must be a nonterminal. The sequence of symbols after the ::= is called the right-hand side (RHS) and terminates with a period. Each "LHS ::= RHS ." is called a production of the BNF. The symbol | means "or" or "alternation". Parentheses may be used for grouping [REID90].

3.3.1 Generating Members of a Language

A member of a language defined by a set of BNF productions is generated by choosing the start symbol (the nonterminal <Declarations> in the above example) and replacing it by one of its possible alternates and then continuing to replace nonterminals on the RHS until the RHS contains nothing but terminals. For example, the declarations

```
VAR I1 : INTEGER ;
VAR R1 , R2 : REAL ;
```

can be derived by beginning with the start symbol <Declarations> and successively replacing the left-most non-terminal by one of its alternate right-hand sides [REID90]:

<Declarations> ::= <Declarations> <Declaration>
::= <Declaration> <Declaration>
::= VAR <Name-Sequence> : <Type> ; <Declaration>
::= VAR <Name> : <Type> ; <Declaration>
::= VAR I1 : <Type> ; <Declaration>
::= VAR I1 : INTEGER ; <Declaration>
::= VAR I1 : INTEGER ; VAR <Name-Sequence> : <Type> ;
::= VAR I1 : INTEGER ; VAR <Name-Sequence> , <Name> : <Type> ;
::= VAR I1 : INTEGER ; VAR <Name> , <Name> : <Type> ;
::= VAR I1 : INTEGER ; VAR R1 , <Name> : <Type> ;
::= VAR I1 : INTEGER ; VAR R1 , R2 : <Type> ;
::= VAR I1 : INTEGER ; VAR R1 , R2 : REAL ;

3.4 Extended Backus Naur Form

Extended BNF (EBNF) adds two new operations to BNF. The first of these is *option* by enclosing a sequence of LHS symbols between square brackets ([and]). The second is *closure* by enclosing a sequence of LHS symbols in braces ({ and }). Option means what is enclosed in brackets occurs either zero or one times. Closure means that what is enclosed in braces occurs zero or more times. We can rewrite the BNF in the previous section by replacing the left recursive definitions with closure:

```
<Declarations>  ::= <Declaration> { <Declaration> } .
<Declaration>   ::= "VAR" <Name> { "," <Name> } ":" <Type> ";".
<Type>          ::= ("INTEGER" | "REAL") .
```

Another example is a translator for simple arithmetic expressions:

```
<Expr>   ::= [ "+" | "-" ] <Term> { ( "+" | "-" ) <Term> } .
<Term>   ::= <Factor> { ("*" | "/") <Factor> } .
<Factor> ::= "(" <Expr> ")" | <Integer> .
```

Thus, an <Expr> is defined as an optional "+" or "-" followed by a <Term> followed by zero or more copies of a "+" or "-" followed by a <Term>. (Giving an English translation can be tough - EBNF is much more precise!)

For more examples of BNF and EBNF, consult various programming language books. Most will present their syntax using some form of BNF [REID90].

3.4.1 EBNF in Relation to the Parsing Process

EBNF notation has a direct relationship to programming language constructs. For instance, option corresponds to an "IF .. THEN" construct, closure to a "WHILE .. DO" construct, while alternation translates to an "IF .. THEN .. {ELSIF ..} ELSE Error" construct. Recall that in recursive descent parsing, there is a single procedure written for each production rule (or EBNF production) in the grammar. This concept is now made clear by looking at the following routine to parse the <Expr> production rule defined above.

```
PROCEDURE ProcExpr ;

(*  <Expr>   ::= [ "+" | "-" ] <Term> { ( "+" | "-" ) <Term> } . *)

BEGIN
   IF Token.Kind IN TokenSet {SY_Plus, SY_Minus} THEN   (*option*)
      IF Token.Kind = SY_Plus THEN GetToken   (* | and "+" *)
      ELSIF Token.Kind = SY_Minus THEN Gettoken   (* "-" *)
      ELSE ReportError
      END;
   END;
   ProcTerm;   (* <Term> *)
   WHILE Token.Kind IN TokenSet {SY_Plus, SY_Minus} DO (*closure*)
      IF Token.Kind = SY_Plus THEN GetToken   (* | and "+" *)
      ELSIF Token.Kind = SY_Minus THEN Gettoken   (* "-" *)
      ELSE ReportError
      END;
      ProcTerm;   (* <Term> *)
   END;
END;   (* ProcExpr *)
```

There are several assumptions here that are not in the TWS and the code that the TWS automatically generates is somewhat different [REID90]. However, the idea is basically the same. The TWS generates source code, such as that above, with one procedure for each EBNF production rule.

3.5 Example Grammar

We now present a slightly modified version of the simple grammar given above in order to illustrate the concepts discussed up to this point. This grammar will be used in future sections as well, to describe semantic actions and attributed grammars.

Simple Grammar:

```
<Expr> ::= [ "+" | "-" ] <Term> { ( "+" | "-" ) <Term> } .
<Term> ::= <Fact> { ( "*" | "/" ) <Fact> } .
<Fact> ::= "(" <Expr> ")" |
           Number |
           Identifier .
```

The nonterminal symbols in this grammar are <Expr>, <Term>, and <Fact>. <Expr> is the start symbol. The terminal symbols are "+", "-", "*", "/", "(", ")", Number, and Identifier.

Here is an sample LL top-down derivation for this grammar. An LL derivation is one in which the left-most nonterminal symbol is always replaced with a right-hand side.

```
<Expr> ::= <Term>
       ::= <Fact> "*" <Fact>
       ::= "(" <Expr> ")" "*" <Fact>
       ::= "(" <Term> "+" <Term> ")" "*" <Fact>
       ::= "(" <Fact> "+" <Term> ")" "*" <Fact>
       ::= "(" Number "+" <Term> ")" "*" <Fact>
       ::= "(" Number "+" <Fact> ")" "*" <Fact>
       ::= "(" Number "+" Identifier ")" "*" <Fact>
       ::= "(" Number "+" Identifier ")" "*" Number
```

Figure 3.3 shows the corresponding derivation tree. The scanner provides the parser with the actual values for Number and Identifier. Figure 3.4 is the recursive descent parser for this simple grammar, as would be generated by the TWS.

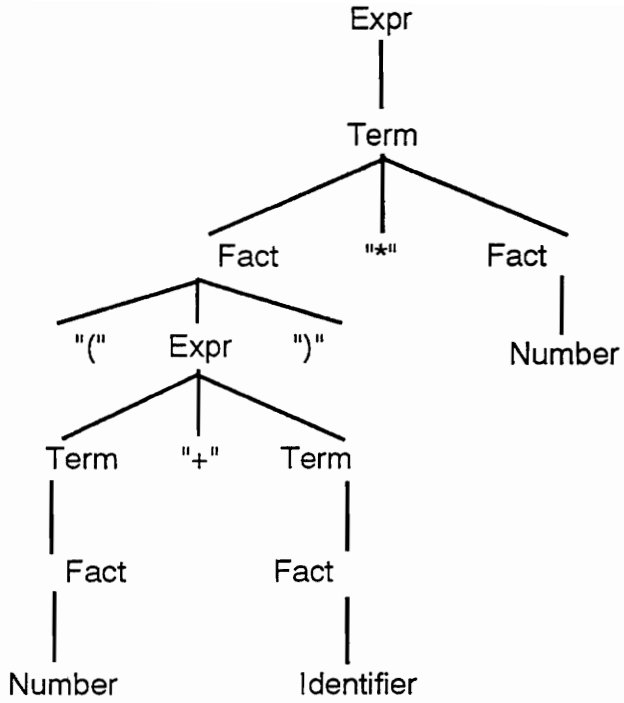


Figure 3.3 Derivation Tree

```
IMPLEMENTATION MODULE G0Parser;
FROM G0Decl IMPORT Token, TokenSet, TokenKind;
FROM TWScan IMPORT GetToken;

PROCEDURE ProcExpr (VAR OK: BOOLEAN);
BEGIN
   IF (Token.Kind IN {SSPlus, SSMinus, SSLParen,
                      TKNumber, TKIdentifier}) THEN
      IF (Token.Kind = SSPlus) THEN
         GetToken;
      ELSIF (Token.Kind = SSMinus) THEN
         GetToken;
      END;  (* IF *)
      ProcTerm (OK);
      WHILE (Token.Kind IN {SSPlus, SSMinus}) DO
         IF (Token.Kind = SSPlus) THEN
            GetToken;
         ELSIF (Token.Kind = SSMinus) THEN
            GetToken;
         END;  (* IF *)
         ProcTerm (OK);
      END;  (* WHILE *)
   ELSE
      OK := FALSE;
   END;  (* IF *)
END ProcExpr;


PROCEDURE ProcTerm (VAR OK: BOOLEAN);
BEGIN
   IF (Token.Kind IN {SSLParen, TKNnumber, TKIdentifier}) THEN
      ProcFact (OK);
      WHILE (Token.Kind IN {SSTimes, SSDivide}) DO
         IF (Token.Kind = SSTimes) THEN
            GetToken;
         ELSIF (Token.Kind = SSDivide) THEN
            GetToken;
         END;  (* IF *)
```

Figure 3.4 Parser for Simple Grammar

```
            ProcFact (OK);
        END;  (* WHILE *)
    ELSE
        OK := FALSE;
    END;  (* IF *)
END ProcTerm;

PROCEDURE ProcFact (VAR OK: BOOLEAN);
BEGIN
    IF (Token.Kind IN {SSLParen, TKNnumber, TKIdentifier}) THEN
        IF (Token.Kind = SSLParen) THEN
            GetToken;
            ProcExpr (OK);
            IF (Token.Kind = SSRParen) THEN
                GetToken;
            ELSE
                OK := FALSE;
            END;  (* IF *)
        ELSIF (Token.Kind = TKNumber) THEN
            GetToken;
        ELSIF (Token.Kind = TKIdentifier) THEN
            GetToken;
        END;  (* IF *)
    ELSE
        OK := FALSE;
    END;  (* IF *)
END ProcFact;

BEGIN
    (* Initialization Code *)
END G0Parser.
```

Figure 3.4 (cont.) Parser for Simple Grammar

3.6 Attributed Grammars and Semantics

Up to now, the format of an EBNF description of a grammar has been given using productions only. However, the TWS allows for more than just production rules. The user may include some TWS directives at the beginning of the EBNF. These directives are specified one per line, and are of the form

%<directive> [value]

where <directive> is a two-character TWS keyword, indicating that the TWS should take some particular action, and [value] is an optional value which may be specified. Table 3.1 lists some of the common TWS directives and explains the meaning. The grammar can also be augmented with symbol attributes and semantic actions, as well defined constants. The constants defined in the EBNF are used by the TWS to generate the scanner/parser declarations for the input grammar. This portion of the input EBNF file takes the form

```
CONST
    <Constant> = <Value>;
        .
        .
        .
    <Constant> = <Value>;
```

where <Constant> is the symbolic name of the constant, like SSPlus, and <Value> is its assigned value, like "+". Additionally, Modula-2 IMPORT statements may be included in the EBNF file, to be imbedded in the generated parser. These import statements appear immediately after the directives, and are needed when the imbedded semantic actions invoke any Modula-2 library routines, or any user-defined procedures. Attributes and semantic actions are explained through this section. Table 3.2 gives the format for the EBNF input file.

Table 3.1 Common TWS Directives

DIRECTIVE	DESCRIPTION
GN	Value is the two-character designator for generated parser. Name of parser will be <value>Parser.
PP	Indicates that the productions should be printed in the list file produced by TWS.
PS	Indicates that the symbol table should be printed in the list file.
PT	Indicates that the syntax tree should be printed in the list file.

Table 3.2 EBNF Input File Format

SECTION	DESCRIPTION
Directives	Begin with "%"; specify properties such as whether or not to print syntax tree, what the first two characters of the name of the generated modules should be, etc.
Import	Modula-2 IMPORT statements to be included in the generated parser.
Attributes	Specify which grammar symbol take on which attributes.
Constant	Specify pre-defined constants for use in the parser and symbol table.
Productions	EBNF production rules for the grammar.

As previously stated, semantics define the meaning of a language. There are two kinds of semantics: *static* and *run-time*. Static semantics refer to specifications which can be tested at compile-time, such as type checking, formal and actual parameter matchups in procedure calls, and using variables before they are declared. Run-time semantics refer to specifications for which testing must wait until run-time, such as numeric range violations, or divide-by-zero errors.

Semantics can be implemented in many ways. The TWS has chosen to implement them as *attributed grammars*. An attributed grammar has a set of attributes, such as value or type, attached to each nonterminal symbol, and a set of semantic rules attached to each production. Attributes allow for information about a nonterminal symbol to be passed up and down the derivation tree during the parsing process. Semantic rules actually perform the semantics of the language (static and run-time) from the attribute values, the symbol table, and the syntax tree.

3.6.1 Attributes

The TWS allows the attributes for nonterminal symbols to be imbedded within the EBNF description of the grammar for the input language. This method provides a description of the language which is programming-language-like. An attribute is a four-tuple of the form

(SYN|INH|LOC) NonTerm "." AttrName "=" DataType ";"

where NonTerm is the nonterminal symbol to which the attribute is being attached, AttrName is the name of the attribute, DataType is the Modula-2 data type for the attribute, and SYN, INH, LOC indicate the attribute type. SYN denotes that the attribute

is *synthesized*, or passed up the tree from child to parent. This is analogous to a procedure passing back the value of a variable to its calling procedure. INH indicates that the attribute is *inherited*, or passed down the syntax tree from parent to child. This is analogous to a procedure calling a subroutine, and passing it the value of a formal parameter. Finally, LOC indicates that the attribute is a *local*, or internal, variable.

Here again is our simple grammar extended to specify attributes for the three nonterminal symbols. These attributes would be appropriate for a simple on-the-fly translator.

```
FROM G0Decl IMPORT Token, TokenSet, TokenKind;
    FROM TWScan IMPORT GetToken;
    ATTRIBUTES
       SYN <Expr>.Val = INTEGER;
       LOC <Expr>.Sign = TokenKind;
       SYN <Term>.Val = INTEGER;
       LOC <Term>.Sign = TokenKind;
       SYN <Fact>.Val = INTEGER;
    <Expr> ::= [ "+" | "-" ] <Term> { ( "+" | "-" ) <Term> } .
    <Term> ::= <Fact> { ( "*" | "/" ) <Fact> } .
    <Fact> ::= "(" <Expr> ")" |
               <Number> |
               <Identifier> .
```

The ATTRIBUTE section of the above EBNF description provides enough information for the TWS to allocate local storage for each attribute, build the formal parameter list for procedure headers (for passing data), and for building the actual parameter lists in procedure calls. Conceptually, the parser replaces the name of each

node of the syntax tree with its attributes. Imbedded semantic actions (discussed in the next section) actually calculate the values for each attribute and pass them up and down the tree. Figure 3.5 depicts the derivation tree for the simple grammar, annotated with the attributes for the nonterminal symbols. Figure 3.6 is the generated parser which utilizes the synthesized attributes for procedure calls and procedure headings, and the local attributes for local variables within each procedure.

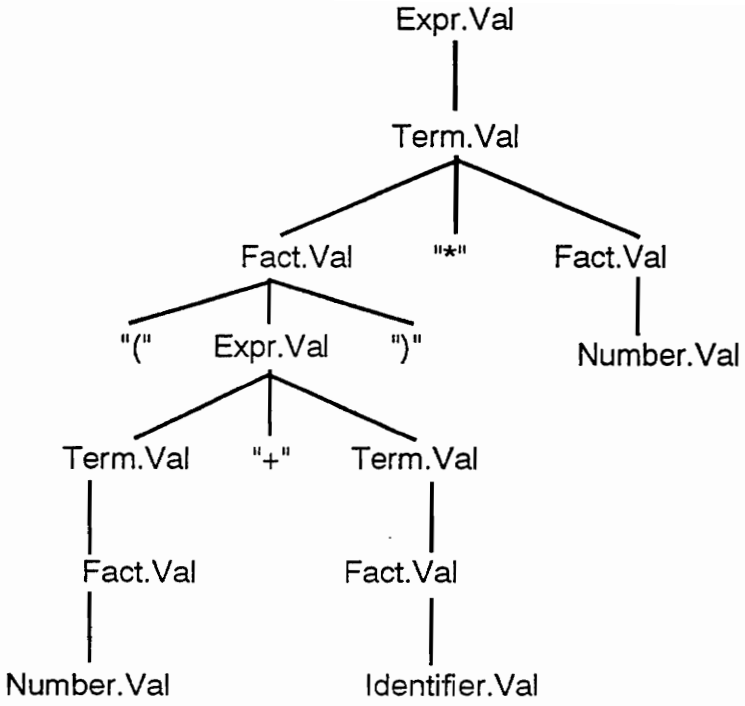


Figure 3.5 Annotated Derivation Tree

```
IMPLEMENTATION MODULE G0Parser;
FROM G0Decl IMPORT Token, TokenSet, TokenKind;
FROM TWScan IMPORT GetToken;

PROCEDURE ProcExpr (VAR Val: INTEGER);
VAR
   TermVal : INTEGER;
   Sign : TokenKind;
BEGIN
   IF (Token.Kind IN {SSPlus, SSMinus, SSLParen,
                      TKNumber, TKIdentifier}) THEN
      IF (Token.Kind = SSPlus) THEN
         GetToken;
      ELSIF (Token.Kind = SSMinus) THEN
         GetToken;
      END;  (* IF *)
      ProcTerm (TermVal);
      WHILE (Token.Kind IN {SSPlus, SSMinus}) DO
         IF (Token.Kind = SSPlus) THEN
            GetToken;
         ELSIF (Token.Kind = SSMinus) THEN
            GetToken;
         END;  (* IF *)
         ProcTerm (TermVal);
      END;  (* WHILE *)
   ELSE
      ParseError (100, Token.ColNo);
   END;  (* IF *)
END ProcExpr;

PROCEDURE ProcTerm (VAR Val: INTEGER);
VAR
   FactVal : INTEGER;
   Sign : TokenKind;
BEGIN
   IF (Token.Kind IN {SSLParen, TKNnumber, TKIdentifier}) THEN
      ProcFact (FactVal);
      WHILE (Token.Kind IN {SSTimes, SSDivide}) DO
```

Figure 3.6 Attributed Parser for Simple Grammar

```
            IF (Token.Kind = SSTimes) THEN
               GetToken;
            ELSIF (Token.Kind = SSDivide) THEN
               GetToken;
            END;  (* IF *)
            ProcFact (FactVal);
         END;  (* WHILE *)
      ELSE
         ParseError (101, Token.ColNo);
      END;  (* IF *)
   END ProcTerm;

   PROCEDURE ProcFact (VAR Val: INTEGER);
   VAR
      ExprVal : INTEGER;
   BEGIN
      IF (Token.Kind IN {SSLParen, TKNnumber, TKIdentifier}) THEN
         IF (Token.Kind = SSLParen) THEN
            GetToken;
            ProcExpr (ExprVal);
            IF (Token.Kind = SSRParen) THEN
               GetToken;
            ELSE
               ParseError (102, Token.ColNo);
            END;  (* IF*)
         ELSIF (Token.Kind = TKNumber) THEN
            GetToken;
         ELSIF (Token.Kind = TKIdentifier) THEN
            GetToken;
         END;  (* IF *)
      ELSE
         ParseError (103, Token.ColNo);
      END;  (* IF *)
   END ProcFact;

   BEGIN
      (* Initialization Code *)
   END G0Parser.
```

Figure 3.6 (cont.) Attributed Parser for Simple Grammar

3.6.2 Semantic Actions

As stated before, there is a set of semantic actions for each production rule. These semantic actions can perform such tasks as mapping attributes to attributes, check the validity of symbols, build and use the symbol table and/or syntax tree, and produce the translation. The TWS semantic actions are merely Modula-2 source code fragments within the EBNF input file. The TWS will place the semantic actions in the proper place in the generated Modula-2 parser module. A semantic action is denoted in the EBNF file by the meta-symbol "#", which indicates the beginning of a semantic action (the "#" is changeable via a TWS directive). All text following the "#", until the end of the line, is treated as a semantic action. These semantic actions may be placed before or behind any symbol (terminal or nonterminal) on the right-hand side of a production. The code fragments represented by these semantic actions are inserted into the generated parser at the point where they appear in the EBNF file. Generally, it is desirable to limit the number of semantic actions in the EBNF file, and use support routines whenever possible. These support routines include the error routines and the code generator. What support routines are needed and how they are implemented depend on the language requirements.

Figure 3.7 is the simple grammar for an on-the-fly interpreter with semantic actions imbedded within the EBNF description. Note the three lines of IMPORT statements at the top of this grammar. The TWS will include these Modula-2 IMPORT directives in the generated parser module.

Figure 3.8 is the resulting parser as would be generated by the TWS, with imbedded semantic actions in boldface. Note that the code fragments have been imbedded at the point in the code corresponding to where they appear in the EBNF file. When the TWS performs its code generation, it traverses a linked list of semantic actions attached to each syntax tree node, and spits each line out into the parser.

```
%GN G0
FROM G0Decl IMPORT Token, TokenSet, TokenKind;
FROM TWScan IMPORT GetToken;
FROM G0Semantics IMPORT IdPtr, GetId;
ATTRIBUTES
   SYN <Expr>.Val = INTEGER;
   LOC <Expr>.Sign = TokenKind;
   SYN <Term>.Val = INTEGER;
   LOC <Term>.Sign = TokenKind;
   SYN <Fact>.Val = INTEGER;
CONST
   SSPlus = "+";
   SSMinus = "-";
   SSTimes = "*";
   SSDivide = "/";
   SSLParen = "(";
   SSRParen = ")";
   TKNumber = <Number>;
   TKIdentifier = <Identifier>;
<Expr> ::=              #Sign := SSPlus;
        [ "+" | "-"    #Sign := SSMinus;
        ] <Term>         #IF Sign = Plus
                       #Val := TermVal;
                       #ELSE Val := -TermVal  END;
        { ( "+"         #Sign := SSPlus
          "-"          #Sign := SSMinus;
        ) <Term>         #IF Sign = SSPlus
                       #THEN Val := Val + TermVal
                       #ELSE Val := Val - TermVal  END;
        .
<Term> ::= <Fact>          #Val := FactVal;
          ( "*"           #Sign := SSTimes;
        | "/"            #Sign := SSDivide;
        ) <Fact>         #IF Sign = SSTimes
                       #THEN Val := Val * FactVal
                       #ELSE Val := Val DIV FactVal  END;
        } .
```

Figure 3.7 Simple Grammar with Semantic Actions

```
<Fact> ::= "(" <Expr>      #Val := ExprVal;
          ")"
          | <Number>      #Val := Token.Val;
          | <Identifier> #GetId(Token.Lexeme, Id);
                         #IF Id = NIL
                         #THEN SemanticError(201)
                         #ELSE Val := Id^.Val  END;
          .
END
```

Figure 3.7 (cont.) Simple Grammar with Semantic Actions

```
IMPLEMENTATION MODULE G0Parser;
   FROM G0Decl IMPORT Token, TokenSet, TokenKind;
   FROM TWScan IMPORT GetToken;
   FROM G0Semantics IMPORT IdPtr, GetId;

   PROCEDURE ProcExpr (VAR Val: INTEGER);
   VAR
      TermVal : INTEGER;
      Sign : TokenKind;
   BEGIN
      IF (Token.Kind IN {SSPlus, SSMinus, SSLParen,
                         TKNumber, TKIdentifier}) THEN
         Sign := SSPlus;
         IF (Token.Kind = SSPlus) THEN GetToken;
         ELSIF (Token.Kind = SSMinus) THEN
            Sign := SSMinus;
            GetToken;
         END;  (* IF *)
         ProcTerm (TermVal);
         IF Sign= SSPlus
         THEN Val := TermVal
         ELSE Val := -TermVal  END;
         WHILE (Token.Kind IN {SSPlus, SSMinus}) DO
            IF (Token.Kind = SSPlus) THEN
               Sign := SSPlus;
               GetToken;
            ELSIF (Token.Kind = SSMinus) THEN
               Sign := SSMinus
               GetToken;
            END;  (* IF *)
            ProcTerm (TermVal);
            IF Sign = SSPlus
            THEN Val := Val + TermVal
            ELSE Val := Val - TermVal  END;
         END;  (* WHILE *)
      ELSE ParseError (100, Token.ColNo);
      END;  (* IF *)
   END ProcExpr;
```

Figure 3.8 Parser for Simple Grammar with Semantic Actions

```
PROCEDURE ProcTerm (VAR Val: INTEGER);
VAR
   FactVal : INTEGER;
   Sign : TokenKind;
BEGIN
   IF (Token.Kind IN {SSLParen, TKNnumber, TKIdentifier}) THEN
      ProcFact (FactVal);
      Val := FactVal;
      WHILE (Token.Kind IN {SSTimes, SSDivide}) DO
         IF (Token.Kind = SSTimes) THEN
            Sign := SSTimes;
            GetToken;
         ELSIF (Token.Kind = SSDivide) THEN
            Sign := SSDivide
            GetToken;
         END;  (* IF *)
         ProcFact (FactVal);
         IF Sign = SSTimes
         THEN Val := Val * FactVal
         ELSE Val := Val DIV FactVal  END;
      END;  (* WHILE *)
   ELSE ParseError (101, Token.ColNo);
   END;  (* IF *)
END ProcTerm;

PROCEDURE ProcFact (VAR Val: INTEGER);
VAR
   ExprVal : INTEGER;
BEGIN
   IF (Token.Kind IN {SSLParen, TKNnumber, TKIdentifier}) THEN
      IF (Token.Kind = SSLParen) THEN
         GetToken;
         ProcExpr (ExprVal);
         Val := ExprVal
         IF (Token.Kind = SSRParen) THEN GetToken;
         ELSE ParseError (102, Token.ColNo);
         END;  (* IF*)
```

Figure 3.8 (cont.) Parser for Simple Grammar with Semantic Actions

```
      ELSIF (Token.Kind = TKNumber) THEN
         Val := Token.Kind
         GetToken;
      ELSIF (Token.Kind = TKIdentifier) THEN
         GetId(Token.Lexeme, Id)
         IF Id = NIL
         THEN SemanticError(201)
         ELSE Val := Id^.Val  END;
         GetToken;
      END;  (* IF *)
   ELSE ParseError (103, Token.ColNo);
   END;  (* IF *)
END ProcFact;

BEGIN  (* Initialization Code *)
END G0Parser.
```

Figure 3.8 (cont.) Parser for Simple Grammar with Semantic Actions

3.7 A Simple Translator

The previous section described a very simple on-the-fly interpreter. The TWS will build the augmented grammar parser module, GOParser, and the parser/scanner interface module, G0Decl. The user must supply other modules. Figure 3.9 depicts the relationship between the major modules for this simple interpreter. Table 3.3 describes the functionality of each of these modules.

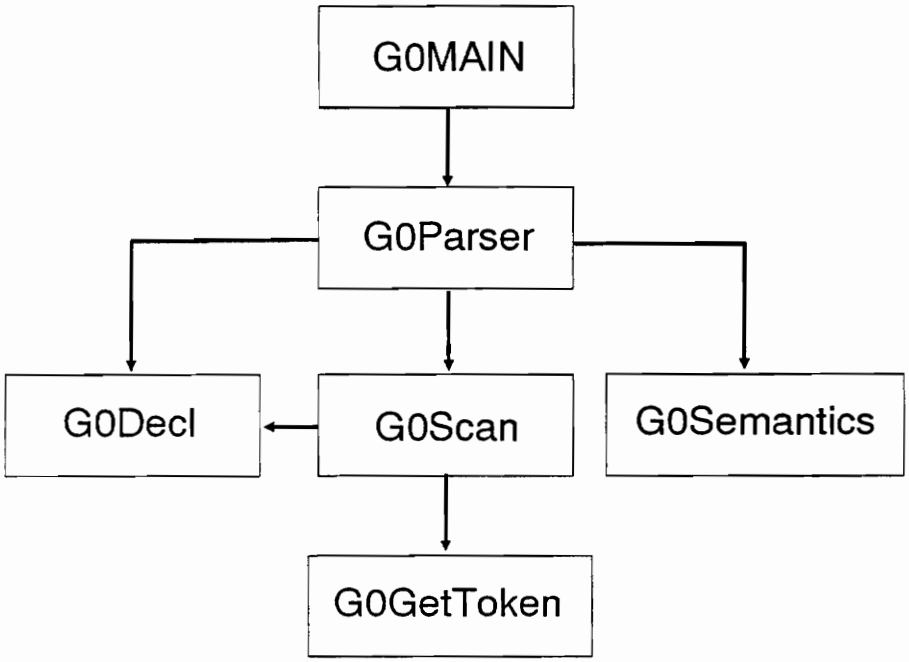


Figure 3.9 Modules for Simple Interpreter

Table 3.3 Module Functions for Simple Interpreter

MODULE	DESCRIPTION
Main	Main driver for the on-the-fly interpreter
G0Parser	Generated parser for the interpreter
G0Scan	Scanner skeleton which interacts with the generated parser, passing it tokens
G0Decl	Scanner/parser declarations generated by TWS
G0GetToken	Routine which scans the input string for the next token when asked by the parser
G0Semantics	Module which contains routines invoked by the imbedded semantic actions

4. TWS System Architecture

The purpose of this section is to explain in detail the inter-workings of the various modules of the TWS. This discussion covers the relationships between the various modules, as well as the primary data structures used by the TWS. This section, combined with the section on the TWS System Overview, comprise the major components of both the User's Guide and the Programmer's Manual.

4.1 TWS Operation

The TWS is a translator which is made by itself. Thus, it has the general architecture of the translators it produces. There are three execution phases. The first phase is to parse the input string, build the abstract syntax tree and symbol table, and perform early semantic analysis. The second phase consists of semantic analysis which checks for useless, inaccessible, and nullable nonterminals. This second phase also includes the calculation of first and follow sets, as well as checking whether or not the grammar is LL(1). The third phase is that of code generation. Figure 2.1 in a previous section depicts this operation of the TWS. Refer to section 2.1, Components of a Translator, for more information on these three phases of a translator.

4.2 TWS Modules

As part of explaining the inter-workings of the different TWS modules, it is necessary to explain exactly what each module is. The TWS consists of some modules which are the same across all possible applications of the software. These modules we

will call template, or *skeleton* modules. The main driver of the TWS, called TWS.MOD, is a prime example of a skeleton module. Another skeleton module is the scanner skeleton, called TWScan. This module is the main scanning routine for the TWS. Its job is to scan the input (the ASCII file containing an attributed grammar description of a language) and pass tokens back to the parser.

Another type of module within the TWS is what is known as a *generated* module. The two generated modules in the TWS are the parser/scanner declaration (or interface) module, TWDecl, and the actual parser itself, TWParser. Because the TWS is itself a translator, the parser module can actually be used to generate the TWS itself. The declaration module TWDecl contains the declarations for the types of symbols found in the target language as specified by the EBNF grammar description. The parser utilizes the declarations found in TWDecl, as does the scanner. In other words, the scanner needs to know which symbols to tokenize, and the parser needs to know just what to do with those symbols. This information is found within the module TWDecl.

Finally, the third type of module in the TWS is that set of modules which are *language specific*. These modules perform such functions as construction of the extended grammar tree, analysis of the input grammar, and code generation. Code optimization is an option which is currently not implemented in the TWS. These routines are utilized by both the attributed parser and the main driver. Specifically, the TWS parser uses these routines to construct the extended grammar tree, and the main driver uses them to perform analysis on the grammar, using the tree as a basis.

Figure 4.1 is a depiction of the three types of modules within the TWS, showing their inter-dependencies as just described. The next sub-section explains some of the language specific modules so that the programmer may gain a greater understanding of the TWS software as a whole.

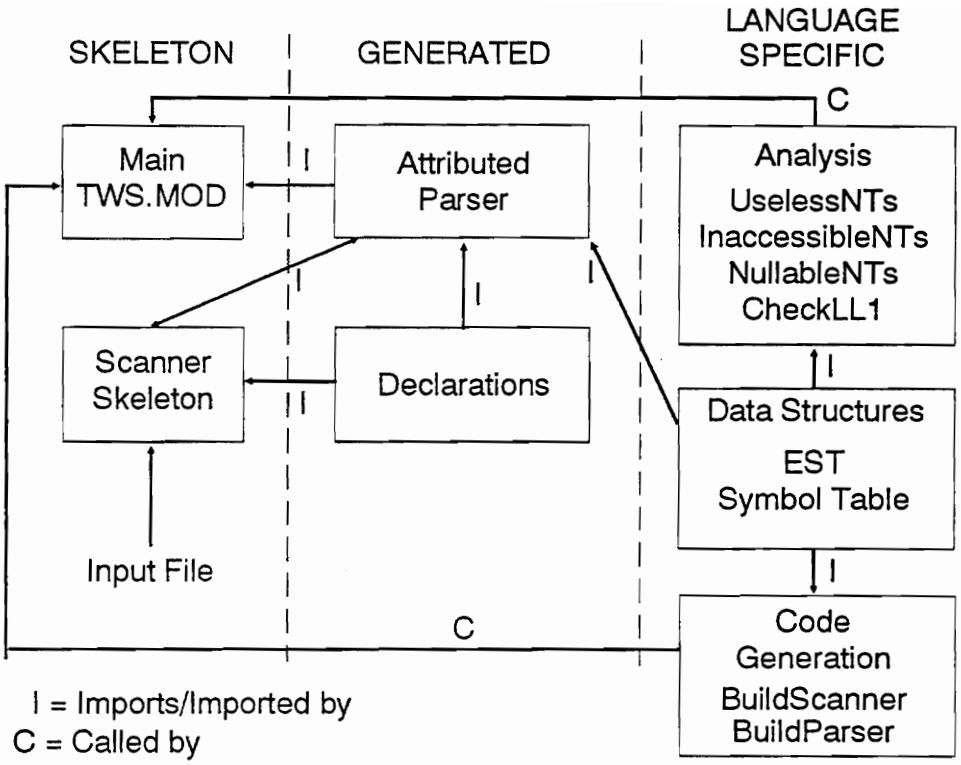


Figure 4.1 TWS Module Interdependencies

4.3 Language Specific Modules

The language specific modules are those modules within the TWS software which relate to grammar analysis, and tree manipulation. The extended grammar tree data structure itself is discussed in another section, and will not be addressed here. The following is a list of the languages specific modules, and a brief description of each.

4.3.1 Module TWESTree

This module contains routines which directly manipulate the extended syntax tree, which is what the TWS calls the extended grammar tree. These routines perform such functions as PrintTree, which prints the contents of the tree on a traversal, and PrintNode, which prints the contents of a particular node in the tree. PrintTree actually calls PrintNode. Module TWESTree also includes a routine called MakeNode, which adds a node to the EST in the proper position, adjusting the tree as necessary.

Other routines include EST analysis procedures. These analysis procedures work directly with the completed EST built by the TWS to perform certain analysis functions on the input grammar. These routines include UselessNonterminals, which searches the grammar via the tree to find useless nonterminals symbols. Useless nonterminal symbols are those that appear on the right-hand side of a production, but not on the left-hand side. In other words, any sentential form which contains a useless nonterminal can never produce a string of terminal symbols. The grammar is therefore in error, and parser generation will not continue.

Two other EST analysis routines are part of module TWESTree and are called InaccessibleNonterminals and NullableNonterminals. InaccessibleNonterminals searches the tree for any nonterminal symbols which appear on the left-hand side of a production,

but not on the right-hand side. In other words, they are inaccessible, and will never be reached in a derivation. NullableNonterminals traverses the tree for those nonterminal symbols which can produce NULL. In either the case of inaccessible or nullable nonterminals, there is no grammar error and parser generation can continue.

The main driver for the TWS, TWS.MOD, calls the extended syntax tree analysis routines, and prints status messages as it goes, and error messages as appropriate.

4.3.2 Module TWSymbol

The EST as implemented by the TWS does not contain all the information for any particular grammar symbol (nonterminal, terminal, identifier, etc.) all within a single node. There is the concept of an EST node, and that of an actual symbol node which is pointed at by the EST node. For example, the EST node might indicate that the grammar symbol it represents is a terminal symbol, but its associated symbol node contains the actual terminal symbol, say the character "a", along with other information about that symbol in particular. This symbol node structure is appropriately called a SymbolNode by the TWS. Module TWSymbol contains routines which relate to these SymbolNode record types. These routines include MakeSymbol, InsertSymbol, and FindSymbol, all of which perform functions implied by their name. MakeSymbol receives information about the specific symbol, and allocates a SymbolNode for it. InsertSymbol will place the created SymbolNode into the EST, associating it with the proper ESTNode. Additionally, it inserts the new SymbolNode into the symbol table, if not already there. It is InsertSymbol which calls MakeSymbol. FindSymbol will find the specified symbol in the symbol table, and as result, in the extended syntax tree.

4.3.3 Module TWSymTab

In addition to keeping grammar specific symbols in symbol nodes, the TWS also keeps track of what it calls the symbol table. This symbol table is nothing more than a linked list of symbol nodes, sorted alphabetically by the symbols themselves. These symbol table nodes are not different nodes from the symbol nodes in the extended syntax tree, but are in fact the same identical nodes. In other words, the symbol table is just another series of threads running throughout the extended syntax tree. This series of threads provides another way of conceptualizing, as well as accessing, the symbols in the input grammar. The primary use, however, of the symbol table is in the generation of the ??Decl definition and implementation modules. As stated previously, the TWSymbol module, when putting a new symbol node into the extended syntax tree, also puts the new node into the symbol table, mainly because inserting the new symbol into the symbol table is part of the job of creating a new symbol node. Module TWSymTab contains routines which actually work with this symbol table, such as OutputSymbolName and SortSymbols, plus procedures which print a symbol listed with all the grammar productions in which it appears. These routines can really be thought of as symbol table support routines.

Additionally, TWSymTab contains an analysis routine called CheckLL1Grammar. This routine checks the input grammar, using the extended syntax tree, to determine whether or not the grammar is LL(1). If it is not, the TWS halts execution, as it will only generate translators for LL(1) grammars. No conversion of the grammar is attempted by the TWS. By rights, this routine should appear within module TWESTree along with the other syntax tree analysis routines. CheckLL1Grammar is also called by the main driver in TWS.MOD. For more information on LL(1) grammars, as well as other elements of language theory, consult the sources cited in the REFERENCES section.

4.3.4 Module TWUtil

This module contains a generic set of utility functions which are specific to the Translator Writer System, like OutputProdNum, PrintProd, and OutputName, to mention a few. These routines perform functions such as string printing and truncation, printing grammar production rules, as well as basic utilities like finding the min and max of two given integers.

4.3.5 Module TWSets

The primary procedure found within module TWSets is CalculateFFSets. This routine traverses the extended syntax tree to calculate the first and follow sets for each symbol in each extended syntax tree. This routine is called also by the main driver TWS.MOD, and the resulting first and follow sets are stored within each EST node for each symbol. The first and follow sets are important for analysis which occurs later in the program, such as checking for LL(1). Additionally, TWSets contains routines CheckStart and Accept, which interact with the generated parser to determine whether a called module within the parser should have even been invoked during the parsing process. These routines are found in Module TWSets because they deal with sets of tokens, as declared in module TWDecl, and scanned by TWScan in order to perform their task.

4.3.6 Module TWCodGen

Module TWCodGen contains the routines which actually go through the extended syntax tree, and generate the source code for the parser, along with the declaration module. As previously explained, there is a direct correlation between EBNF meta-symbols and programming language constructs. For example, the closure symbol ({}) corresponds to a WHILE loop, and option (|) corresponds to an IF construct. Module TWCodGen is able to use this association to generate the parser for the input grammar. The routine BuildParser traverses the extended syntax tree, and writes out the proper source code, with semantic actions, to the output parser file, based on the type of node, and the symbol attached to it. Semantic actions are stored in a semantic action file. If a symbol has any semantic actions associated with it, the BuildParser routine reads the specified number of characters from the semantic action file, and prints them to the output parser file. The semantic action file is created during the scanning process. When they are found, they are immediately written out to this file. When the symbol being parsed is placed into the extended syntax tree, the beginning and ending bytes (within this file) of its semantic actions are stored in the ESTNode structure. Additionally, formal parameters for procedure headings are also generated, based on the attributes specified in the input grammar.

4.3.7 Module TWTypDef

Module TWTypDef contains the major type definitions for the TWS, particularly those of the ESTNode record, the SymbolNode record, and the attribute and import record types. These type definitions are combined into this single module, rather than have each data structure declared within the module dealing with it, for the purpose of clarity, as

well as conciseness in the IMPORT section of each of the other TWS modules. Figure 4.2 is a listing of these type definitions which are at the heart of the Translator Writer System. A detailed description of these data structures and their uses are contained in the section on TWS data structures.

```
DEFINITION MODULE TWTypDef;
FROM TWDecl     IMPORT TokenKind, LexemeRange, LexemeType;

CONST
   MaxSymbols  = 254;   (* Max number of symbol table entries *)
   MaxProductions = 250;  (* Max number of productions allowed *)
   MaxESTNodes    = 250;  (* Max number of EST Node entries *)

TYPE
   ProdKind = (PKNEBNF, PKProd, PKExpression, PKTerm, PKSymbol,
               PKAttributes, PKConsts, PKImport);
   ESTKinds    = (ESTProd, ESTNonterminal, ESTSpecSymbol,
                  ESTIdentifier, ESTAlt, ESTClosure,
                  ESTConcat, ESTOption);
   ProdNum     = [0..MaxProductions];
   ProdSet     = SET OF ProdNum;
   SymbolNum   = [0..MaxSymbols+1];
   SymbolSet   = SET  OF SymbolNum;
   SymbolPtrs  = POINTER TO SymbolNode;
   AttribPtrs  = POINTER TO AttributeNode;
   ImpPtrs     = POINTER TO ImpNode;
   ESTNodePtr  = POINTER TO ESTNode;
   ESTNode = RECORD
                NodeNum       : CARDINAL;
                Kind          : ESTKinds;
                SymbolPtr     : SymbolPtrs;
                Left          : ESTNodePtr;
                Right         : ESTNodePtr;
                First         : SymbolSet;
                Follow        : SymbolSet;
                PrintSymbol   : BOOLEAN;
                LHSUsed       : ESTNodePtr;
                RHSUsed       : ESTNodePtr;
                FirstBeginSA  : CARDINAL;
                FirstEndSA    : CARDINAL;
                FollowBeginSA : CARDINAL;
                FollowEndSA   : CARDINAL;
```

Figure 4.2 TWS Definition Module TWTypDef.DEF

```
            ProdNumber    : ProdSet;
            Nullable      : BOOLEAN;
          END;
    SymbolNode = RECORD
                 Next       : SymbolPtrs;
                 Kind       : TokenKind;
                 Lexeme     : LexemeType;
                 Len        : LexemeRange;
                 LHSUsed    : ESTNodePtr;
                 RHSUsed    : ESTNodePtr;
                 Num        : SymbolNum;
                 Attributes : AttribPtrs;
                 Enumeration : LexemeType;
               END;
    AttributeNode = RECORD
                    Next       : AttribPtrs;
                    Name       : LexemeType;
                    DataType   : LexemeType;
                    AttribType : LexemeType;
                  END;
    ImpNode = RECORD
              Name      : LexemeType;
              Visited   : BOOLEAN;
              NextImp   : ImpPtrs;
              NextMod   : ImpPtrs;
              FromField : BOOLEAN;
            END;

END TWTypDef .
```

Figure 4.2 (cont.) TWS Definition Module TWTypDef.DEF

4.3.8 Module TWScan

This module, TWScan.MOD, comprises the generic scanner skeleton mentioned above. As previously stated, the main purpose of the scanner is to scan the input string, and pass a token back to the parser when asked. This task is performed by the routine called GetToken. Thus, TWScan interacts very heavily with module TWParser, with the concept being that the parser, when ready, asks the scanner for a token, and the scanner provides the next token in the input string to the parser. The parser then takes the appropriate action based on the token it receives. Module TWScan also includes a routine called PrintToken, which is called to print the token just found by the scanner out to the listfile produced by the TWS.

4.3.9 Module TWInOut

This module contains generic TWS input/output routines.

4.3.10 Module TWSupprt

This module contains support routines useful for debugging purposes, character string manipulation, and type conversion.

Modules TWInOut and TWSupprt do not require much exposition here, as the brief descriptions just provided basically tell the whole story of these modules. A programmer wishing to know more of these generic modules need only look briefly at the Modula-2 code itself.

4.4 System Flow

Having described the individual modules, template, generated, and language specific, it is necessary at this point to present a high-level system flow description of the TWS. We have just seen the purpose and function of the individual modules, and most of their inner routines, as well as the inter-workings between them; however, in order to provide a complete picture, this section will demonstrate the general flow of processing within the TWS system. In this section, the reader will see how it all fits together from a very high level.

4.4.1 Main Driver

The main driver first of all begins with prompting the user for the name of the input file. This file is the ASCII file containing the EBNF description of the input grammar. It then opens the files for output (listfile or console), and another to hold the semantic actions. It then calls GetToken to get the first token in the input string, and calls the parser. The job of the parser is to build the extended syntax tree and symbol table from the input grammar. Upon returning from the parser, the main driver then calls SortSymbols to lexicographically sort the symbols in the completed symbol table. Following this, some information is output which may be of interest to the user, such as an ordered listing of the productions, as well as a tabular listing of the symbols in the input grammar.

Four tree analysis routines are then invoked. These analysis routines, as previously stated, act upon the extended syntax tree built by the parser. The main driver calls UselessNonterminals, InaccessibleNonterminals, and NullableNonterminals. These routines have been explained above. It then invokes CalculateFFSets to calculate the first

and follow sets for each symbol in the complete extended syntax trees. This set of trees (one for each production) is then printed out in a readable format.

Finally, the routine CheckLL1Grammar is called. If it returns to the main driver that the grammar is NOT LL(1), processing is halted. If the grammar IS LL(1), the main driver then calls BuildScanner, followed by BuildParser, to produce the generated modules.

4.4.2 Parser

The parser has the major tasks of constructing the extended syntax tree, and the symbol table. The parser is a recursive descent parser, that is it has one procedure for each EBNF production rule in the grammar. As stated before, since the TWS is itself a translator, the parser itself has been generated and rehosted onto the TWS. Basically, the action of the parser is that it looks at the token it currently has received from the scanner, and makes the appropriate procedure call based on the token's type. For example, if the token is a nonterminal "NT", the parser then calls the parser routine ProcNT. If the token is an identifier, it calls ProcIdentifier, and so on. This parsing method is easily understood by examining the simple grammar example in the TWS overview section. In that section, the G0 simple grammar was gradually augmented with attributes and semantic actions. By looking carefully at the productions in that grammar, then looking at the resulting recursive descent parser for that language, the parsing algorithm is quickly grasped. Additionally, the token received from the scanner is placed into an ESTNode and put in the proper position in the current extended syntax tree. If the token is a symbol in the grammar itself, a SymbolNode will also be created for it, attached to the ESTNode, and added to the symbol table. In this way are these two data structures constructed. Figure 4.3 shows the general system flow just described above.

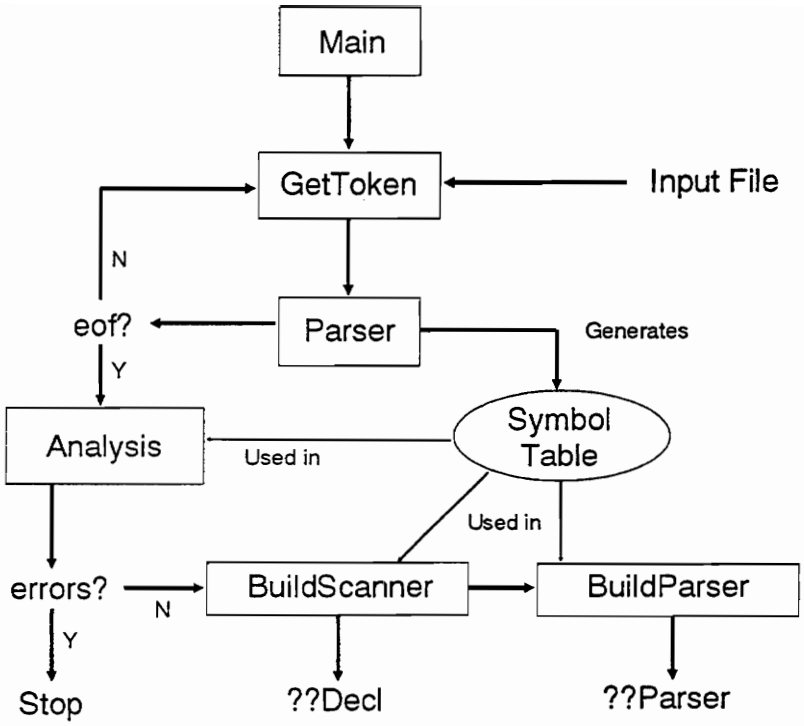


Figure 4.3 TWS General System Flow

5. TWS Data Structures

The Translator Writer System has been described in various stages, moving in the direction from general to specific. The principles behind the TWS have been discussed, as well as the interaction between the various modules and routines of the TWS. References have also been made concerning the major data structures of the software. This section describes those structures in detail, and relates them to the process of generating the parser for the input grammar's language. This will help the programmer gain a much greater understanding of the internal workings of the TWS.

5.1 Extended Syntax Tree

Up to this point, the extended syntax tree (or abstract syntax tree) has been discussed only enough to understand the material being presented currently. By now, it should be obvious that the extended syntax tree is the primary data structure of the TWS. This tree has been referred to as the *extended syntax tree*, the *extended grammar tree*, the *abstract syntax tree*, and just the *syntax tree*. All these names have been used synonymously, as they refer to the exact same tree structure within the TWS. This tree structure is hereafter referred to as the extended syntax tree, since that is what the TWS code itself calls it.

5.1.1 Definition

An Extended Syntax Tree (EST) is a method of representing the structure of a language. Conceptually, it is a tree whose root node is the left hand side of a production,

and whose children are the right hand side of a production. Therefore, there is an extended syntax tree for each production rule. The EST is the primary technique used by the Translator Writer System to represent the productions in the input grammar. To illustrate, Figure 5.1 lists the simple grammar again.

```
<Expr> ::= [ "+" | "-" ] <Term> { ( "+" | "-" ) <Term> } .
<Term> ::= <Fact> { ( "*" | "/" ) <Fact> } .
<Fact> ::= "(" <Expr> ")" |
           Number |
           Identifier .
```

Figure 5.1 Simple Grammar

In Figure 5.2, the extended syntax tree is shown for the nonterminal symbol <Expr>. Note that, as alluded to before, some of the nodes in the tree contain grammar symbols, while some of them contain EBNF meta-symbols. This all-inclusive nature of the tree is what allows for the direct translation from an EBNF grammar description to Modula-2 programming language constructs. While Figure 5.2 does not illustrate the actual implementation of the EST by the TWS, it does demonstrate how an extended syntax tree is developed.

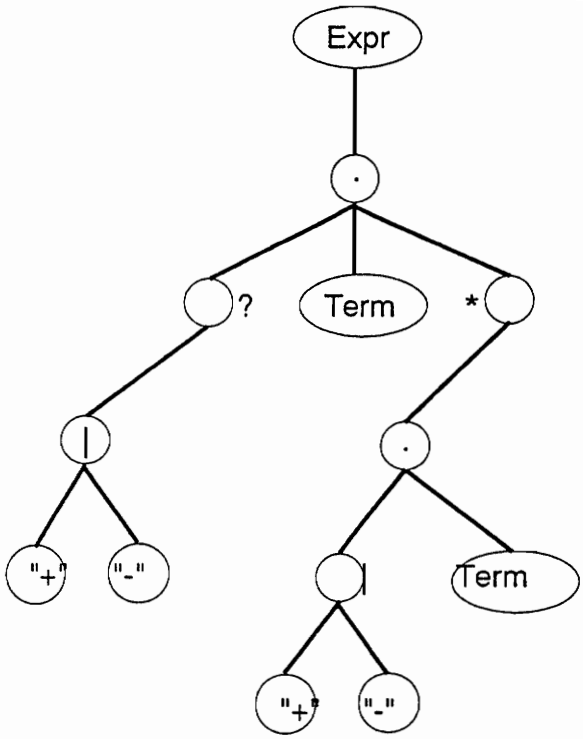


Figure 5.2 Extended Syntax Tree for <Expr>

5.1.2 Implementation

Because the extended syntax tree can have an undetermined number of children per node, there must be a different way of representing this tree in the Modula-2 programming language. This is because Modula-2 does not allow for open-ended arrays of pointers, like the C programming language. Therefore, the TWS implements the extended syntax tree as a binary tree. Thus, a concatenation node with three children (three symbols being concatenated) is represented as one node having two children, with its left child being another concatenation node. This left child's children are the first two symbols being concatenated. The right child of the concatenation node is the third of the original symbols being concatenated. Figure 5.3 represents such a concatenation as it would be depicted in an extended syntax tree. Figure 5.4 shows the TWS implementation of the same structure. In both these figures, and in all others depicting extended syntax trees, the following symbology is used for representing EBNF meta-symbols: "." for concatenation; "|" for alternation; "?" for option; and "*" for closure.

Having explained this structure, Figure 5.2 depicted the extended syntax tree for the nonterminal <Expr> production. Figure 5.5 now depicts that same extended syntax tree as would be implemented by the TWS.

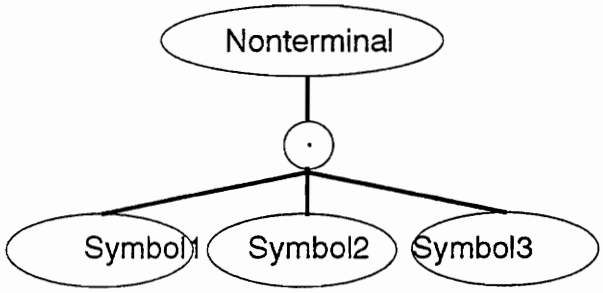


Figure 5.3 EST for Concatenation

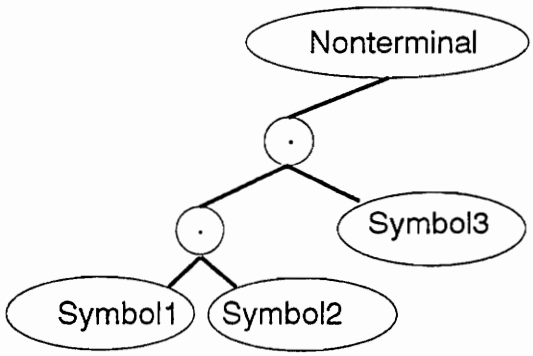


Figure 5.4 TWS Implementation of Concatenation

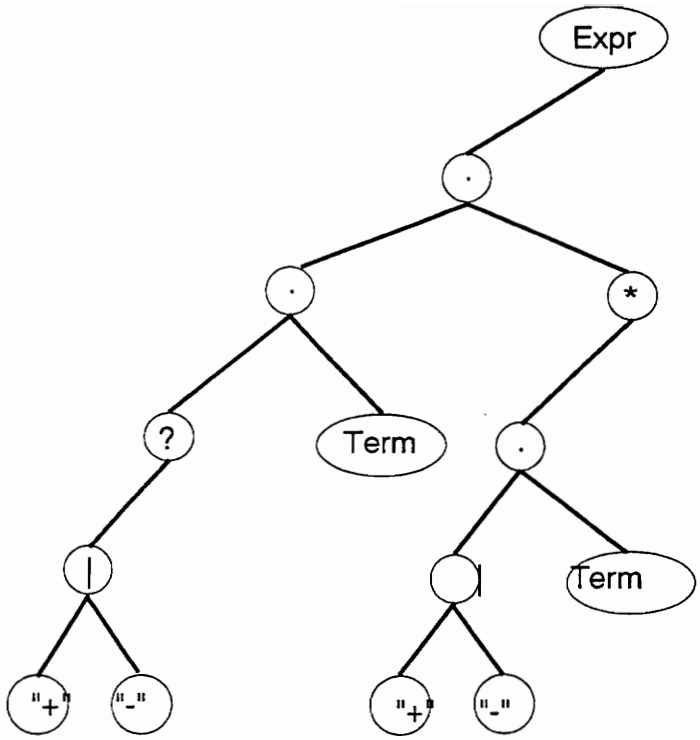


Figure 5.5 TWS EST for <Expr>

5.1.3 EST Record Types

As previously stated, the EST as implemented by the TWS does not contain all the information for any particular grammar symbol (nonterminal, terminal, identifier, etc.) all within a single node, as depicted in Figure 5.5. Instead, there is an EST node, which contains more generic information about either the symbol or meta-symbol, and an actual symbol node which is pointed at by the EST node. In particular, an EST node contains information pertaining to the EBNF description of the input grammar, such as whether or not the current token scanned is an option or a closure. It also may contain information if the token is a nonterminal symbol, for instance. In such a case, the EST node would indicate that the symbol is a nonterminal, but would not contain the actual symbol itself, like <Term>. That information would be kept in a symbol node. The EST node contains a pointer to the symbol node. Another example would be the EST node indicating that the grammar symbol it represents is a terminal symbol, but its associated symbol node contains the actual terminal symbol, say the character "a", along with other information about that symbol in particular. This symbol node structure is appropriately called a SymbolNode by the TWS. In the case where the EST node was for an EBNF meta-symbol, like concatenation, the symbol node pointer is NULL. In this way, the two concepts, that of symbols and meta-symbols, are separated from each other, providing for a more object-oriented approach, instead of keeping all information for a symbol in a single node.

Having explained the EST node structure, Figure 5.6 illustrates the nonterminal <Expr> extended syntax tree down to the data structure level, as implemented by the TWS.

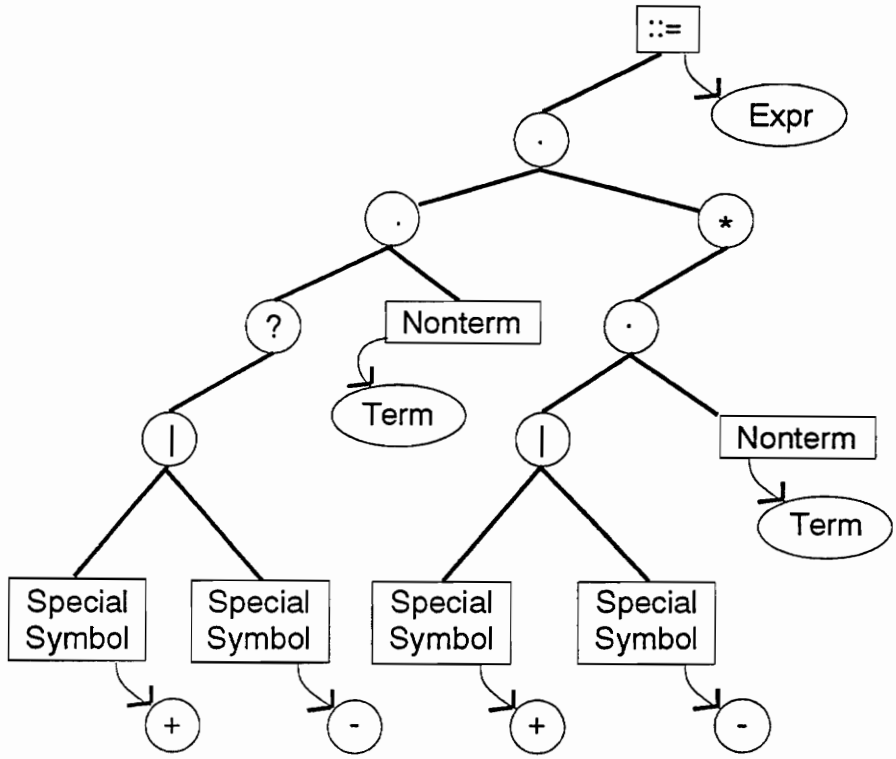


Figure 5.6 TWS EST for <Expr> shown as Data Structures

5.1.4 EST Record Definitions

Figure 4.2 (Section 4.3) was a listing of the TWS module TWTypDef. This module contains, as the name implies, the type definitions for the TWS implementation of the extended syntax tree. Table 5.1 shows the major fields in the ESTNode record, along with their purpose and/or function. Table 5.2 depicts the same information for the SymbolNode.

There is an additional record type utilized by the TWS for the extended syntax tree. It has been shown that the input grammar may attach attributes to symbols in the grammar. These attributes are of three types: inherited, synthesized, or local, as explained in the Section 3.6.1. The attribute record type is the method by which the TWS relates the input grammar symbols to their attributes, if any. Within each SymbolNode, there is a pointer to an attribute node. Actually, it is a pointer to the head of a linked list of attributes, as a symbol may have more than one attribute. Thus, there is a single attribute node per attribute per symbol. The concept of the attribute node is yet another level of encapsulation for the representation of the input grammar. Again, keeping the attributes in a separate structure from both the EST node and the symbol nodes makes the extended syntax tree more manageable, as well as readable. Table 5.3 shows the AttributeNode fields, along with their purpose and/or function.

Table 5.1 ESTNode Record Fields

NAME	DESCRIPTION
NodeNum	Ordinal number of the EST node as it would be visited in a preorder traversal of the tree.
Kind	The kind of node within the EST this record represents; i.e. terminal, option, closure, etc.
SymbolPtr	A pointer to the record type which actually holds the EST symbol being referenced (SymbolNode).
Left	The left child of the node.
Right	The right child of the node.
First	The first set for the particular symbol pointed at by SymbolPtr.
Follow	The follow set for the particular symbol pointed at by SymbolPtr.
PrintSymbol	Flag to indicate whether or not symbol should be printed out in a traversal of the tree.
LHSUsed	Pointer to the next occurrence of the particular symbol on the left-hand side of a production.
RHSUsed	Pointer to the next occurrence of the particular symbol on the right-hand side of a production.
ProdNumber	Number of the production from the EBNF file from which this node is generated.
Nullable	TRUE is the symbol is a nonterminal, and is nullable; false otherwise.

Table 5.2 SymbolNode Record Fields

NAME	DESCRIPTION
Next	Pointer to the next symbol in the symbol table.
Kind	Kind of symbol; i.e. identifier, constant, integer, etc.
Lexeme	Character representation of the symbol; i.e. symbol name.
Len	Length of the symbol name.
LHSUsed	Pointer to the next occurrence of the particular symbol on the left-hand side of a production.
RHSUsed	Pointer to the next occurrence of the particular symbol on the right-hand side of a production.
Num	Number of the symbol in the symbol table.
Attributes	Pointer to the first record holding information on the attributes of the particular symbol.

Table 5.3 AttributeNode Record Fields

NAME	DESCRIPTION
Next	Pointer to the next attribute node for the particular symbol.
Name	Name of the attribute.
DataType	Data type of the attribute; i.e. real, integer, character, etc.
AttribType	Type of attribute; i.e. inherited, synthesized, or local.

One other data structure not included in TWTypDef is an array of pointers to EST nodes. This array is called EBNFArray, since it is simply an array holding, in order of appearance in the EBNF file, pointers to each of the root nodes of the extended syntax trees. In other words, it is really an array of extended syntax trees.

5.2 Use of the Extended Syntax Tree

The extended syntax tree is a dynamic data structure, created on the fly as the TWS parses the input. The various field values are placed within each node, the symbol nodes attached to the EST nodes, and attributes attached to the symbol nodes. All this processing is done at parse time. Once the syntax tree has been successfully built, the analysis routines are called from the main driver program, such as calculating first and follow sets. These analysis routines utilize the completed extended syntax tree to find inaccessible nonterminals, useless nonterminals, and nullable nonterminals. Finally, another pass is made through the tree for code generation.

An inaccessible nonterminal is one which appears on the left-hand side of a production, but not on the right-hand side. In other words, it can never appear in a sentential form of the language, since it can never be reached in a derivation. The only exception to this rule is the start symbol. The TWS considers the nonterminal on the left-hand side of the very first production to appear in the EBNF to be the start symbol. Looking at it from the point of view of the extended syntax tree, the search for inaccessible nonterminals in the grammar is really the search for any root nodes whose symbol is not attached to any internal or leaf nodes of any tree. The algorithm is simple: scan the array of extended syntax trees, and put the root node symbols in a set of left-hand side nonterminals; then start with another set of nonterminals just containing the start symbol; search each tree and add all internal and leaf nonterminals to this set;

subtract the second set from the first set, and the result is a set of nonterminals on left-hand sides of productions, but not on right-hand sides. In short, a set of inaccessible nonterminals.

Similarly, the algorithm for useless nonterminals works much the same way as that for inaccessible ones. First, find all internal or leaf nodes pointing to nonterminal symbols nodes. While traversing, add these nonterminals to a set. This set, by definition, should exclude root nodes, or left-hand side nonterminals. Finally, create a set of just left-hand side nonterminals. Subtracting the second set from the first leaves a set of useless nonterminals; i.e. nonterminals which appear on the right-hand side of a production, but not on a left-hand side.

After analysis routines are done on the grammar using the tree, the main driver then calls the routine to calculate first and follow sets. Again, these algorithms utilize the completed extended syntax tree. It is easy to traverse the various trees to trace a particular nonterminal, and determine each of these sets. Once the first and follow sets are computed, the main driver invokes CheckLL1Grammar, which in turn uses the extended syntax trees, specifically the first and follow sets for each symbol. As before, the structure of the tree, and the recursive nature of all these algorithms go well together. For information on first and follow sets, and LL(1) grammars, consult the REFERENCES section.

Finally, if the grammar has been verified to be LL(1), the main driver invokes the routine for generating the parser source code. This routine is called BuildParser, and is contained in TWS module TWCodGen. It is in this routine that the correlation between EBNF meta-symbols and programming language constructs becomes of value. It has been stated that a recursive descent parser, which is what the TWS generates, is one in which there is a single separate procedure for each production nonterminal. By a simple preorder traversal of the extended syntax tree, BuildParser knows exactly what Modula-2 source code to write out, and at what time. For example, if the EST node currently being

visited is a closure node, then BuildParser puts out "WHILE (Token.Kind IN SymbolSet{...}) DO", where ... is the set of symbols and meta-symbols the parser expects to find as children of the closure node. The same principle holds true for alternation and option. Although there is some scanning ahead in the traversal of the tree during code generation, in order to determine which symbols to place within the WHILE condition for example, this basic preorder traversal of the extended syntax tree is primarily what drives the code generation process. Figure 5.7 lists the parser generated for the simple grammar shown in Figure 5.1. This parser was listed in a previous section, but is repeated here to make referring to it easier. How this parser would be generated by the TWS should be clearer now that the extended syntax tree is understood.

```
IMPLEMENTATION MODULE G0Parser;
FROM G0Decl IMPORT Token, TokenSet, TokenKind;
FROM TWScan IMPORT GetToken;


PROCEDURE ProcExpr (VAR OK: BOOLEAN);
BEGIN
   IF (Token.Kind IN {SSPlus, SSMinus, SSLParen,
                      TKNumber, TKIdentifier}) THEN
      IF (Token.Kind = SSPlus) THEN
         GetToken;
      ELSIF (Token.Kind = SSMinus) THEN
         GetToken;
      END;  (* IF *)
      ProcTerm (OK);
      WHILE (Token.Kind IN {SSPlus, SSMinus}) DO
         IF (Token.Kind = SSPlus) THEN
            GetToken;
         ELSIF (Token.Kind = SSMinus) THEN
            GetToken;
         END;  (* IF *)
         ProcTerm (OK);
      END;  (* WHILE *)
   ELSE
      OK := FALSE;
   END;  (* IF *)
END ProcExpr;


PROCEDURE ProcTerm (VAR OK: BOOLEAN);
BEGIN
   IF (Token.Kind IN {SSLParen, TKNnumber, TKIdentifier}) THEN
      ProcFact (OK);
      WHILE (Token.Kind IN {SSTimes, SSDivide}) DO
         IF (Token.Kind = SSTimes) THEN
            GetToken;
         ELSIF (Token.Kind = SSDivide) THEN
            GetToken;
```

Figure 5.7 Generated Parser for Simple Grammar

```
            END;  (* IF *)
            ProcFact (OK);
         END;  (* WHILE *)
      ELSE
         OK := FALSE;
      END;  (* IF *)
   END ProcTerm;


   PROCEDURE ProcFact (VAR OK: BOOLEAN);
   BEGIN
      IF (Token.Kind IN {SSLParen, TKNnumber, TKIdentifier}) THEN
         IF (Token.Kind = SSLParen) THEN
            GetToken;
            ProcExpr (OK);
            IF (Token.Kind = SSRParen) THEN
               GetToken;
            ELSE
               OK := FALSE;
            END;  (* IF*)
         ELSIF (Token.Kind = TKNumber) THEN
            GetToken;
         ELSIF (Token.Kind = TKIdentifier) THEN
            GetToken;
         END;  (* IF *)
      ELSE
         OK := FALSE;
      END;  (* IF *)
   END ProcFact;

   BEGIN
      (* Initialization Code *)
   END G0Parser.
```

Figure 5.7 (cont.) Generated Parser for Simple Grammar

5.3 Symbol Table

So far, the generation of the parser by the TWS has been explained, showing the major data structures required for this code generation process. The resulting parser is called ??Parser, where ?? is some two character description of the language. However, there is also another module created by the Translator Writer System, that of the scanner. Actually, the scanner exists as a skeleton as part of the baseline TWS. However, the declarations must be produced so the scanner will know exactly what to scan for when asked for a token. The declarations for the generated parser are found in module ??Scan.DEF and ??Scan.MOD, where ?? has the same meaning as it did with the parser. This section explains how the declarations are generated, and as in the case of the parser, will do so by explaining first the data structure required, then how that data structure is used.

5.3.1 Symbol Table and Implementation

A typical computer language has a declaration section where the attributes of constants, types, and variables are defined, followed by a block which defines what is to be done. It would be inefficient (at best) to use attributes to save the declaration information until it is used. Thus, the declaration module for the target language is generated by the TWS, to be used by the parser and scanner.

As stated, the skeleton scanner must know exactly what target language tokens to scan for when asked by the parser. To do this, it must know what symbols are part of the language. This is the idea behind generating the ??Decl modules. These modules define the symbols to be scanned for in the target language, to be used by the target language translator. The data structure implemented by the TWS for the generation of

??Decl is that of the symbol table. Very simply, it is an array of symbols, a symbol being simply a character string representing that symbol, such as "A" or "+". However, the actual implementation of the symbol table by the TWS is a little more complex. When a new symbol node is attached to an EST node in the extended syntax tree, this new symbol node has all the information needed about the particular. Any array of symbols, like the symbol table, would lose much by just being an array of character strings. Therefore, the TWS uses the newly inserted symbol node to represent the symbol also in the symbol table. Therefore, in actuality, the symbol table in the TWS is an array of symbol node pointers. Keep in mind that the symbol table is used to generate the declarations for the ??Decl module output by the TWS. This ??Decl generated module is then later used by the generated ??Parser module and the scanner for the target language translator. Remember to separate the concepts of the TWS symbol table from the target language symbol table.

In addition to the array of symbol pointers, the TWS also keeps track of the symbol table from within the extended syntax tree. Within this tree, there is a series of threads going from some symbol nodes to others. Basically, it is a linked list of symbol nodes. However, this linked list is lexicographically sorted. The head of this list is a symbol node pointer called SymbolTable. When a new symbol is inserted into the extended syntax tree, the TWS does a linear search of that linked list, and adjust the links so that the new symbol node appears in this symbol table linked list in the correct position. If it is a new symbol, another pointer is added to the array, called SymbolArray, described above, which then points to this new symbol node in the tree. Thus, there are two ways of conceptualizing the symbol table within the TWS: that of a linked list of symbol nodes in the extended syntax tree; and that of an array of pointers to symbol nodes.

5.3.2 Use of the Symbol Table

Figure 5.8 is a listing of a sample declaration definition module, as might be generated by a sample run of the TWS using the simple grammar. The symbol table is the primary data structure behind the generation of the scanner declarations. In fact, the symbol array itself almost drives the generation of these modules. Much of the definition module for ??Decl is a template, in that certain structures like TokenType are always the same. However, there is a type declaration in this definition module called TokenKind. This is the primary portion of this module which is generated. As shown in Figure 5.8, TokenType consist of a number of ordinal representations of various symbols one might expect to find in an EBNF grammar description. These values in this set are nothing more than the positions in SymbolArray at which these symbols can be found.

As one might suspect, there are certain predefined constants, such as SSLParen, SSPlus, and SSDivide to name a few, which make this module more easily read and understood. These constants, are again, the positions in the SymbolArray at which would appear "(", "+", and "/", or whatever the symbol happens to be. The remaining values in this TokenType set are simply indices into the SymbolArray for symbols which have no predefined position. These symbols, when encountered by the scanner/parser, are represented in this set as "TK" followed by a one-up increment of the current highest position in the SymbolArray. In Figure 5.8, there are exactly ten of them. In this example, they represent the ten single digits. This section of the scanner declarations is thus completed by procedure BuildScanner by stepping through the array, and adding the ordinal value of each symbol used in the grammar to this set.

```
DEFINITION MODULE G0Decl;

FROM G0GetChr IMPORT LineRange;

CONST
   LexemeSize = 20;
   SSSize =  1;
   KWSize =  0;
TYPE
   TokenKind = (SSLParen, SSRParen, SSTimes, SSPlus,
                SSMinus, SSDivide, TK44, TK45, TK46, TK47,
                TK48, TK49, TK50, TK51, TK52, TK53,
                TKNone, TKEOFz);
   KWRange = [0..KWSize];
   KWString = ARRAY KWRange OF CHAR;
   SSRange = [0..SSSize];
   SSString = ARRAY SSRange OF CHAR;
   SymbolTableS  = ARRAY [SSLParen..TK53] OF SSString;
   LexemeRange = [0..LexemeSize];
   LexemeType = ARRAY LexemeRange OF CHAR;
   TokenSet = SET OF TokenKind;
   TokenType = RECORD
                  Kind : TokenKind;
                  Col : LineRange;
                  Len : LexemeRange;
                  Lexeme : LexemeType;
                  CardVal : LONGCARD;
                  RealVal : LONGREAL;
                  ActionPos : CARDINAL;
                  ActionLen : CARDINAL;
               END;
VAR
   SymbolTable : SymbolTableS;
   KWStart, KWEnd, SSStart, SSEnd : TokenKind;
   Token : TokenType;
END G0Decl .
```

Figure 5.8 Definition Module G0Decl.DEF

The implementation module ??Decl.MOD generated by the TWS, is much simpler to produce. It uses the array of symbol node pointers, SymbolArray, to initialize the values in the array SymbolTable, by just stepping through the array and setting the value of the symbols to the value of the Lexeme field in each symbol node. Figure 5.9 is a listing of this module. As seen from the listing, the values for those symbols which have predefined constants are initialized, as well as the generic TK ordinals. Two other constants are initialized: SSStart and SSEnd. SSStart is an index to the first symbol in the SymbolTable array, and SSEnd is the index into the last symbol.

```
IMPLEMENTATION MODULE G0Decl;
BEGIN
   SSStart := SSLParen;
   SSEnd   := TK53;
   SymbolTable[SSLParen] := "(";
   SymbolTable[SSRParen] := ")";
   SymbolTable[SSTimes] := "*";
   SymbolTable[SSPlus] := "+";
   SymbolTable[SSMinus] := "-";
   SymbolTable[SSDivide] := "/";
   SymbolTable[TK44] := "0";
   SymbolTable[TK45] := "1";
   SymbolTable[TK46] := "2";
   SymbolTable[TK47] := "3";
   SymbolTable[TK48] := "4";
   SymbolTable[TK49] := "5";
   SymbolTable[TK50] := "6";
   SymbolTable[TK51] := "7";
   SymbolTable[TK52] := "8";
   SymbolTable[TK53] := "9";
END G0Decl .
```

Figure 5.9 Implementation Module G0Decl.MOD

6. TWS Example

The previous sections have explained the Translator Writer System at several levels. First, there was a brief discussion on translation of computer languages. Following this was a general overview of the TWS. These two sections provided a high-level understanding of the concepts behind the TWS. Third, a description of the architecture of the TWS was presented, with an explanation of each individual module, as well as the inter-workings between them and their routines. Section 5 covered the major data structures of the TWS. That section defined the extended syntax tree, showed how this structure was implemented by the TWS, and how it was actually used to generate the parser. Additionally, that section defined and showed the implementation and use of the symbol table to generate the scanner declarations. This section now attempts to bring all these pieces together so as to provide an idea of how the Translator Writer System may be used.

6.1 Implementing a New Language

The primary need filled by the TWS is for on-the-fly translators, which produce the output concurrently with parsing the input. These translator can be command interpreters such as BASIC, or structure to structure interpreters, like the UNIX utilities "grep" and "sed". The capability of the TWS to support imbedded attributes and semantic actions within the EBNF file provides a simple means of allowing the value of symbols to be computed and used while the input string is actually being parsed.

The TWS itself is a language which actually describes translators. Therefore, if one were to run the TWS against its own grammar specification, the result would be the parser and scanner declarations for the TWS language. This is because the code

generation routines have been invoked by the main routine. However, not all uses of the TWS require that source code be generated. For example, let us assume the existence of a new language, called L0, to be used for an interactive command interpreter. We have already seen how the TWS generates a parser and the scanner declarations for L0, but of what use is it? Obviously, the parser should be usable, if it was produced. This is where the skeleton, generated, and language specific modules, described in Section 4.2, come into play. There must be a main driver module, L0Main, for this interpreter. A scanner, L0Scan, must also exist. These two modules can be derived from the TWS main driver and scanner. Hence, their designation as skeleton modules; they can and should be modified to suit the particular application, in this case a command line interpreter. For example, since this interpreter will translate the input interactively, there is no need to call the BuildScanner and BuildParser routines, since no code is to be generated. There is also no need to call the analysis routines for useless, inaccessible, and nullable nonterminals.

The language specific modules would contain any routines needed by the L0 interpreter. These routines could include mathematical procedures, or perhaps a procedure to list a directory for example. Additionally, this modules would contain the application-specific procedures invoked from the parser via the semantic actions imbedded within the EBNF for L0. There is no need for any of the TWS analysis routines for useless, inaccessible, and nullable nonterminal symbols, or to check for LL(1). Additionally, there is no need to generate an extended syntax tree. This tree was built and used by the TWS to generate the L0 parser, but since no source code is output by L0, this data structure can be eliminated. This module could be called L0Semantics.

Finally, the generated modules for L0 are the parser and the scanner declarations. These and all the other modules would be compiled, then the main driver linked to create an executable. Since the EBNF file contained semantic actions which performed certain operations at certain points, the parser actually does some processing of the input, as

specified. An example of this might be when the token "LISTDIR" is received from the scanner, the parser calls the procedure ListDir which then gives a directory listing of the current disk. In this way, the input is scanned, parsed, and the results are immediately available. Thus, the parser is also an interactive command line interpreter for the L0 language. Figure 6.1 shows a generic description of an application for which the TWS was used to generate the parser/interpreter. The optional portions of this generic application are also noted in the figure.

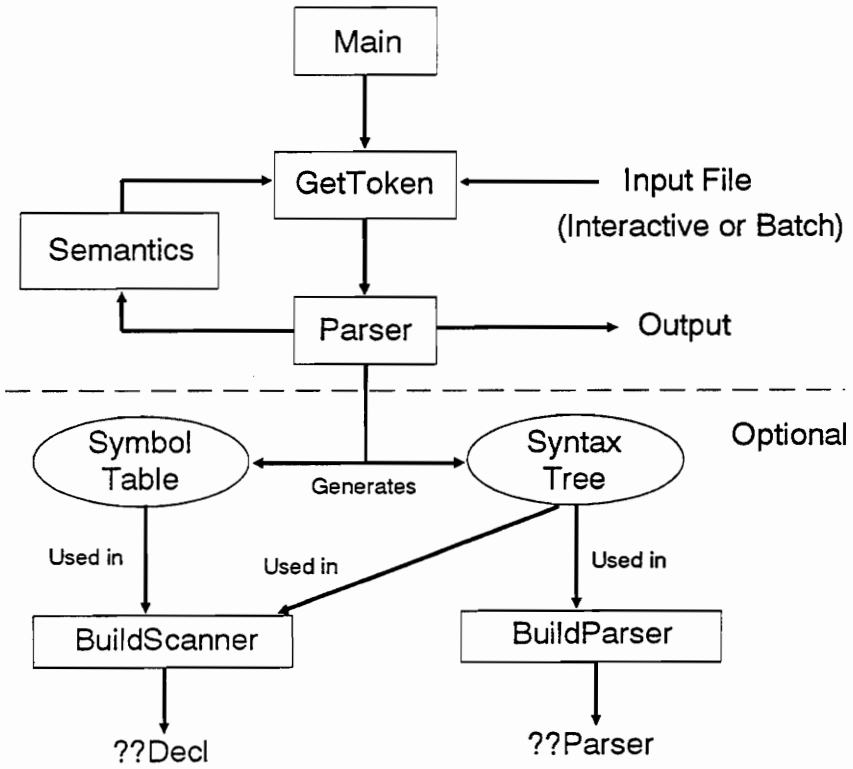


Figure 6.1 Generic TWS-Based Application Flow

6.2 Hand Calculator Overview

The hand calculator is used in this section as an example of how to use the TWS software. This specific example is like the generic one described above for L0, in that the hand calculator is essentially a command line interpreter. The purpose of this section is NOT to explain how the hand calculator itself works, in terms of syntax and execution, but how to GENERATE the hand calculator interpreter using the TWS. The hand calculator software is, like the TWS itself, an example of a translator. It is provided with the TWS software. It consists of Modula-2 definition and implementation modules for a simple hand calculator with memory. The purpose of the hand calculator is to interactively perform mathematical functions, compute expressions, and store and recall variables and their values in memory. The modules all have filenames of the form CA*.MOD and CA*.DEF. There are the usual support routines as found in the TW* modules, as well as some modules specific to the calculator "language". These are the language specific modules described in the previous section, and in Section 4.2. Included in this set are such modules as CAErrors, CAGetChr, CAParSup (parser support routines), and CASymb, to name a few.

The skeleton modules are the main driver CALC.MOD, and the skeleton scanner, CAScan.MOD. These modules can be copied from the analogous TWS modules, and modified according to the needs of the particular application. These modifications are relatively minor. They include specifying the proper IMPORT statements at the beginning of each module, and changing the call to the parser routine in the main driver to invoke the proper procedure. This procedure is called ProcLang, where "Lang" is the name of the start symbol in the hand calculator grammar. Remember that a recursive descent parser has a single procedure for each EBNF production. Therefore, the first parse routine to call is that of the start symbol. The modules to be generated are the parser, CAParser, and the scanner declarations, CADecl. In this case, however, the

capability of the TWS to support imbedded semantic actions within the EBNF is exploited to its fullest potential. The result is essentially a full-scale interpreter, much like the imaginary L0 described above. Figure 6.2 shows the relationship between each major module of the hand calculator, and the data structures used by each. Table 6.1 defines what each of these modules are.

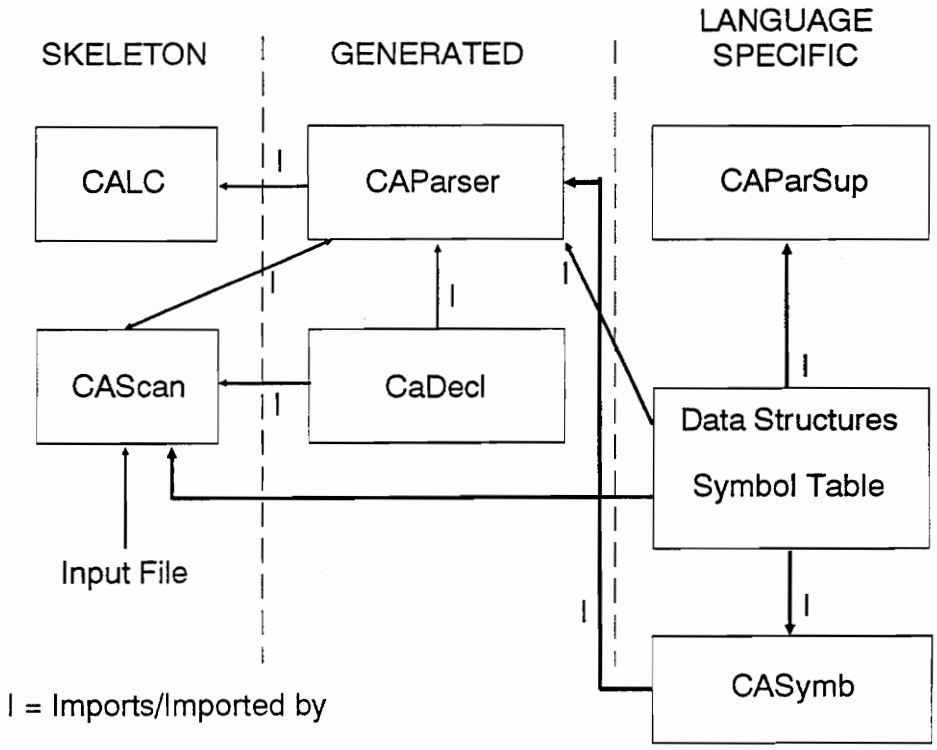


Figure 6.2 Hand Calculator Module Interaction

Table 6.1 Hand Calculator Module Descriptions

NAME	TYPE	DESCRIPTION
CALC	Skeleton	Main driver for the hand calculator.
CAScan	Skeleton	Hand calculator scanner module.
CAParser	Generated	Parser generated by the TWS for the hand calculator.
CADecl	Generated	Scanner/parser declarations created by the TWS.
CASymb	Language-dependent	Module containing routines to perform operations specified by semantic actions.
CAParSup	Language-dependent	Module containing routines needed by the parsing process itself.

6.3 Hand Calculator EBNF

Figure 6.3 is a listing of the file CALC.BNF, containing that EBNF description. This listing is provided for an understanding of the hand calculator language syntax. The semantic actions have been excluded from the listing, to make it more readable. The semantic actions deal with attributes of symbols, values assigned to variables, or certain processing steps to take depending on the symbol received from the user input. As explained previously, these semantic actions, or code fragments, are imbedded within the generated parser at the point the appear within the EBNF description of the grammar, but are just not shown here.

```
%GN CA
%PP
%PS
FROM InOut      IMPORT Write, WriteString, WriteLine, WriteLn,
                       WriteLongInt;
FROM TWSupprt   IMPORT EqualStr;
FROM RealInOut IMPORT WriteLongReal;
FROM CADecl     IMPORT Token, TokenSet, TokenKind, LexemeType;
FROM CAScan     IMPORT GetToken;
FROM CASymb     IMPORT GetSymbTab, PutSymbTab, SymbFound,
                       InsertSymb, Compute, SymbTabRec, SymbTabPtr,
                       MakeReal, PrintTable, PrintSymbol,
                       PrintNumber;
FROM CAParSup   IMPORT ProdKind, PrintProd, UndentPrintProd,
                       CheckStart, Accept;
FROM LMathLib0 IMPORT sqrt, exp, ln, sin, cos, arctan;

ATTRIBUTES
   LOC <Stmt> . Symb = SymbTabPtr;
   LOC <Stmt> . Rec = SymbTabRec;
   LOC <Stmt> . SaveIt = BOOLEAN;
   SYN <Expr> . Rec = SymbTabRec;
   LOC <Expr> . Opr = CHAR;
   LOC <Expr> . Sign = CHAR;
   SYN <Term> . Rec = SymbTabRec;
   LOC <Term> . Opr = CHAR;
   SYN <Fact> . Rec = SymbTabRec;
   LOC <Fact> . Base = SymbTabRec;
   LOC <Fact> . Symb = SymbTabPtr;

CONST
   SSExclam = "!";
   SSLparen = "(";
   SSRparen = ")";
   SSAssign = ":=";
   SSPlus = "+";
   SSMinus = "-";
```

Figure 6.3 EBNF Description of Hand Calculator

```
    SSDivide = "/";
    SSMult = "*";
    SSDelimit = ";";
    SSComma = ",";
    TKReal = <Real>;
    TKInteger = <Integer>;
    TKIdentifier = <Identifier>;
    KWPrtTbl = printtable;
    KWSvTbl = savetable;
    KWQuit = quit;
    KWsqrt = sqrt;
    KWexp = exp;
    KWln = ln;
    KWsin = sin;
    KWcos = cos;
    KWarctan = arctan;
    KWpower = power;

    <Lang> ::= { ( printtable | savetable | <Stmt> ) ";" } quit .

    <Stmt> ::= [ "!" ] <Expr> [ ":=" <Expr> ] .

    <Expr> ::= [ ( "+" | "-" ) ] <Term> { ( "+" | "-" ) <Term> } .

    <Term> ::= <Fact> { ( "*" | "/" )<Fact> } .

    <Fact> ::= "(" <Expr> ")"
             | <Real>
             | <Integer>
             | <Identifier>
             | sqrt "(" <Expr> ")"
             | exp "("<Expr> ")"
             | ln "(" <Expr> ")"
             | sin "(" <Expr> ")"
             | cos "(" <Expr> ")"
             | arctan "(" <Expr> ")"
             | power "(" <Expr> "," <Expr> ")"  .
END
```

Figure 6.3 (cont.) EBNF Description of Hand Calculator

6.4 Hand Calculator Generation

To generate to hand calculator is very simple. The skeleton and support modules are already in place, having been taken from the analogous TWS modules and modified appropriately. The missing modules are the parser and the scanner declarations.

Run the TWS. When asked for the name of the input file, the file CALC.BNF should be specified. The software will ask if the listing should be displayed on the console or not. If not, a file CALC.LST is created, with such information as a list of the productions, the symbol table, and a printout of the extended syntax tree. After the analysis routines are performed on the grammar using the syntax tree, the scanner declarations and parser are generated by BuildScanner and BuildParser respectively. The scanner declarations and parser are named CADecl and CAParser, respectively. There is a definition and implementation module for each.

After separately compiling CADecl and CAParser, as well as the skeleton and language dependent modules (CAParSup, CAErrors, etc.), link CALC.MOD to create CALC.EXE. CALC.EXE is the hand calculator: a parser, semantic analyzer, and interpreter all in one, built automatically from the EBNF file. Running CALC.EXE gives a prompt on the screen at which the user types in the input string, possibly some mathematical expression. The hand calculator parser (CAParser) parses that input string, and performs the operations and procedure calls specified by the semantic actions for each token. For example, when the parser receives the divide symbol "/", it actually performs the division on those values (once both the dividend and divisor are known), and prints the results. This scheme holds true for all other input into the hand calculator: the input is scanned, and the appropriate semantic action code is executed according to the token(s) recieved. The output appears on the screen, and the prompt is again displayed, until the user types "quit".

6.5 Summary

The same process described above can be used to generate any other interpreter, like BASIC, or to generate intermediate code for some high-level language. The only limits are that the produced parser ??Parser cannot exceed the maximum number of characters allowable by the particular Modula-2 compiler being used. The FST public domain Modula-2 compiler has a limit of 64K characters.

7. Discussion

This section summarizes and discusses the work done on the TWS during the course of this project. The idea behind this project was to evaluate the major data structures of the TWS, particularly the extended syntax tree, and propose a more abstract data type design. The extended syntax tree is such a complex structure that much time and effort is required to fully understand it.

7.1 Analysis of the Extended Syntax Tree

The first step in this project was to fully investigate the TWS software, and come to a total understanding of the extended syntax tree data structure. This understanding includes what it is, how it is built, and how it is used. This was a prerequisite to understanding how the TWS works overall. There were many test runs made of the TWS, with WriteLn statements at key places so as to see just what was happening at each and every step. The code was scrutinized painstakingly. Once the tree structure, as well as the symbol table, was conceptually understood, two approaches were investigated to redesign the tree structure.

7.1.1 Object-oriented Approach

First, the possibility of breaking up the major fields within the ESTNode record type was examined. This would have involved separating those fields with deal with common elements in totally new record types. Some fields would have been combined with the SymbolNode record fields to create an entirely different set of data structures.

Each of these records would have contained not only the information about its fields, but also the operations, or procedures, used to manipulate these fields. Hence, an object-oriented approach, whrer the various node types were the objects, and the fields and procedures were the attributes and actions. This approach was abandoned, however, due to the limitation of the Modula-2 programming language. An object-oriented language like C++ is much better suited for this kind of redesign.

7.1.2 Hierarchical Approach

The next approach examined was to maintain the overall structure of the extended syntax tree, but redesign the EST nodes so that they were comprised of nothing but pointers to other record types. These record types would separate out of the ESTNode those fields which were directly related to each other. For example, an ESTNode would have contained a pointer to a symbol node like it does now, a pointer to a node containing first and follow sets, and a pointer to a node containing semantic action information, to name a few. These new nodes themselves would also encapsulate data within additional record types, and contain pointer to these records. Hence, a hierarchical approach to the redesign. This approach also proved to be undesirable, because of the recursive nature of many of the TWS routines. It would have added a level of complexity to the tree, without adding any value to the implementation.

7.1.3 Approach Taken

The extended syntax tree remains much as it was when delivered. the only redesign which was feasible had already been done, that of implementing the tree as a binary tree. Prior to this version, the extended syntax tree represented multiple children of a node via a child/sibling pointer scheme. For example, Figure 7.1 show a simple three-symbol concatenation as it is currently implemented by the TWS. Figure 7.2 shows the tree as originally implemented. The binary tree structure lends itself to traversal, analysis, and manipulation.

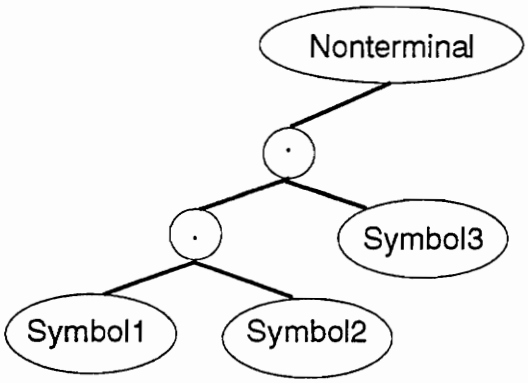


Figure 7.1 Current TWS Tree Implementation

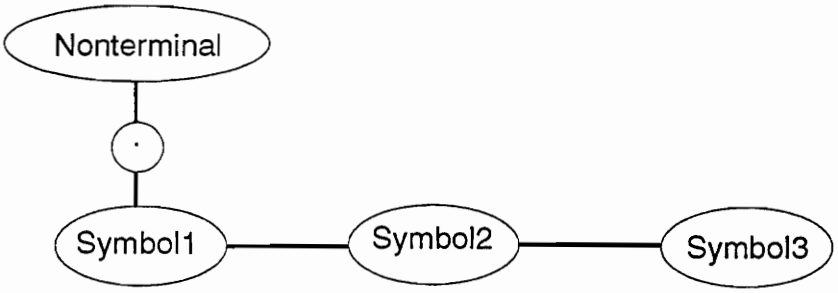


Figure 7.2 Original TWS Tree Implementation

7.2 Module Analysis

The separation of areas of concern described above for the data structures has been implemented in the TWS modules. The original TWS had two modules, TWParSup, and TWSymb, which together contained all the routines for construction, and maintenance of both the extended syntax tree and the symbol table. The tree analysis routines were also in these modules. Additionally, they controlled the maintenance of symbol nodes, and contained the type definitions for all the major data structures. These modules were essentially eliminated. They were replaced by the modules TWTypDef, TWESTree, TWSymbol, TWSets, TWSymTab, and TWutil. These modules have already been explained in Section 4.3, Language Specific Modules.

All the import statements in every TWS module have been cleaned up so that no module imports any routine, variable, or structure that it does not need. Also, no module declares a routine, variable, or structure that is not used either by itself or by another module. This cleaning up of the Modula-2 code applies to both the definition and implementation modules.

7.3 Lessons Learned

Perhaps the most enjoyable and educational aspect of this project is the generation of this documentation. To be able to document the functionality and internals of the TWS required a great amount of time and energy. This is because of the extremely complicated nature of the software, not just in terms of the abstract data types involved, but because it required some background in language theory and compiler design. This paper is a complete explanation of the TWS, how it works, and how to use it. A thorough description of each module is provided, along with a detailed account of the

interactions between their routines. Additionally, this report fully documents the major data structures, and gives a practical example of how to use the software for a real application. This report is included as part of the TWS baseline.

7.3.1 Technical Knowledge Gained

In addition to a more solid understanding of the parsing process, as well as the entire translation process for computer languages, the TWS provides some information in the area of language theory and compiler design. For example, does the TWS increase productivity? While there are other more conventional methods of producing parser and translators, the TWS does this automatically by constructing the extended syntax tree from the input grammar. The parser/translator is then produced from the syntax tree. This method works for two reasons: 1.) the ability to translate EBNF to the extended syntax tree, then do analysis on the tree to show that the input grammar is LL(1); and 2.) the direct correlation between EBNF meta-symbols and programming language constructs. A simple preorder traversal of the syntax tree yields the basic flow of the parser code. Imbedded semantic actions perform whatever tasks are desired based on the current token. Thus, the methodology of the TWS is simple to understand, while its actual implementation is complicated.

7.4 Conclusion

The Translator Write System is a valuable piece of software to anyone working in the area of translators, compilers, and/or interpreters. It is also an excellent educational tool, in that students can see the parsing process actually implemented, as well as gain

a greater understanding of translators in general. Once the student grasps the power of the TWS language to generate translators, the software can be used to create practical applications which require some sort of parsing capability. The hand calculator is good example of a practical application which is really a command-line interpreter. This report has shown how attributes and semantic actions were included in the EBNF input to make this possible, as well as how all the pieces were put together in the creation of this simple interactive utility.

REFERENCES

AHO73 Aho, Alfred V.; Ullman Jeffrey D.; *The Theory of Parsing, Translation and Compiling.* Englewood Cliffs: Prentice-Hall, 1973.

AHO77 Aho, Alfred V.; Ullman Jeffrey D.; *Principles of Compiler Design.* Reading: Addison-Wesley, 1977.

BARR79 Barrett, William A.; Bates, Rodney M.; Gustafson, David A.; Couch, John D.; *Compiler Construction: Theory and Practice.* Chicago: Science Research Associates, 1979.

DAV81 Davie, A. J. T; Morrison, R.; *Recursive Descent Compiling.* New York: John Wiley and Sons, 1981

HOP79 Hopcroft, John E.; Ullman, Jeffrey D.; *Introduction to Automata Theory, Languages, and Computation.* Reading: Addison-Wesley, 1979.

LEW76 Lewis, P. M.; Rosenkrantz, D. J.; Stearns, R. E.; *Compiler Design Theory.* Reading: Addison-Wesley, 1976.

REID90 Reid, Thomas F.; *Translator Writer System*, Documentation.

VITA

Stuart A. Odom was born on January 22, 1964 in Fairfax, Virginia, and graduated from Robert E. Lee High School in Springfield, Virginia in 1982. Stuart graduated from George Mason University in Fairfax, Virginia in 1986 with a Bachelor of Science degree in Computer Science. He began in September 1988 working toward his Master of Science degree in Computer Science at Virginia Tech.

After graduating from George Mason, Stuart worked for ROH, Incorporated for one-and-a-half years. His primary area of responsibility was development and maintenance of database applications on a mini-computer system.

In April 1988, Stuart joined General Electric. He was a UNIX systems programmer until May 1992. Currently, he is a Software Engineer in the Management and Data Systems Division of GE Aerospace.