A User Definable Language Interface for Forth

T. A. Ivanco and G. Hunter
York University Institute for Space and Terrestrial Science
Toronto, Ontario
CANADA

Abstract

The Forth programming language has often been criticized as a *write-only language*, i.e., a difficult language to read due mainly to the postfix syntax and the implied operands on the stack. In this article we describe two meta compiler implementations written in Forth-79 and Forth-83 respectively that enhance the language so that it supports a user defined syntax and semantics. The language is defined in conventional Backus-Naur notation with semantics written in Forth embedded within the definitions.

Introduction

The Forth compiler uses a simple postfix syntax whose words are separated by "white-space." The words are the functional entities of the language. Data, in whatever form, is typically passed to these functions on a stack. Named entities such as variables and constants are organized in a tree structured vocabulary. Access is not controlled by ALGOL scope rules, but rather through a path search mechanism similar to that used to access files in the UNIX programming environment.

In our view, these characteristics of Forth together form an error prone programming environment. A programmer easily loses sight of a function's operands; they are no longer explicit entities, but rather implied locations on a stack. Forth programs contain a number of stack manipulation functions (e.g., **DUP**, **OVER**) that are incidental to the programming task, thus obscuring the underlying algorithm. Lisp, like Forth, is a difficult language to read; Lisp has a prefix syntax and Forth a post-fix syntax. However, Lisp, unlike Forth, has the merit that parameters of a function are passed explicitly.

On the other hand, Forth provides an extremely compact, efficient and extensible programming environment. Our purpose was to design and implement a meta compiler for a user defined language that would address the aforementioned difficulties. This environment provides a mechanism for a programmer to create new language and data structures; building up Forth to the application, rather than cutting the application down to suit Forth.

The meta compiler can also be used to generate new compilers, and also application programs such as data base inquiry parsers. The data base primitives could be written using standard Forth definitions and the function generated by the metacompiler would parse a request as specified by the input language definition and call specific functions as a result of the input language semantics. It is possible to write a pre-processor that accepts an input source, and generates Forth engine code or the native code for the processor. Other applications include the enhancement of the existing Forth compiler; for example, with an infix to postfix expression analyzer. The meta compiler could construct an "immediate" execution word that parses the incoming definition and inserts the appropriate Forth function addresses into the dictionary.

This article describes two approaches to implementing a meta compiler, one using Forth-79, and the other meta compiler, Forth-83; the latter method utilizes more sophisticated mechanisms. Both methods will accept a language definition written in Backus-Naur form. The meta compiler will generate a new function, whose purpose is to accept a source input in the given language and to execute the appropriate semantic actions associated with the given constructs of the language.

The first method constructs a new Forth word which is the compiler that implements the given language definition using a recursive descent technique.

The second method does not create a new Forth word as the compiler. It uses a common parser and semantic interpreter combined with parser/semantic tables that together, implement the compiler for a given language definition. In this case, the compiler kernel and the tables contain the syntactic and semantic elements of the new language.

The two methods (based upon a recursive descent or table driven system) will accept the source input and generate new Forth words as the semantics of the language definitions dictate. The compiled programs are directly executed by the Forth engine without any further processing by the Forth compiler.

Notation

In order to generate the new compiler, it is first necessary to describe the syntax and semantics of the language. This list of definitions is used by the meta compiler to create the target compiler.

The language is described by a set of production rules. A production rule is a definition of a non-terminal symbol and is made up of a left-hand-part and right-hand-part. The left-hand-part is the name of the non-terminal symbol and the right-hand-part consists of a series of terminal and nonterminal symbols.

The non-terminal symbol represents a phrase in the grammar (e.g., **<STATEMENT>**) and a terminal symbol represents a single lexical entity typically returned by the lexical analyzer (e.g., **IF**). It may be helpful to consider the non-terminal symbol as a function and the terminal symbol as an action within that function.

A non-terminal definition can contain several alternative right hand parts. Each alternative is either a completely defined rule each with the same goal symbol, or one production rule with each alternative separated by an "|". As a simple example of parsing consider:

```
<ifstatement>  ::=   IF <condition>  THEN <statement> |
                     IF <condition>  THEN <statement>
                                     ELSE <statement>
<statement>   ::=    EXIT | ADD
<condition>   ::=    NOT <condition>   |  ZERO
```

This definition describes an **<ifstatement>** as a series of symbols starting with **IF**, followed by some **<condition>**, a **THEN**, followed by another **<statement>**, and finally an optional **ELSE <statement>**.

Given the previous definitions, consider the following language string:

```
IF NOT ZERO THEN ADD ELSE EXIT
```

The leading lexical symbol **IF** indicates that it is necessary to parse an **<ifstatement>** and that this requires that **<condition>** be considered. At this point we have:

```
IF <condition> ...
```

A **<condition>** can be either of the lexical entities **NOT** or **ZERO**. The language string symbol **NOT** indicates the use of the first alternative for **<condition>**. The parser accepts **NOT** and

recursively calls for a parse of **<condition>** which results in the recognition of **ZERO**. At this point the parser will unravel back to the initial call of **<condition>** and accept **THEN**. The next symbol **ADD** is a **<statement>**. As the following symbol is **ELSE** the parse would continue with the alternate path in which another **<statement>** is expected. The symbol **EXIT** satisfies the second statement and the parse is complete.

BNF describes a rule with an indefinite number of like elements or phrases through recursion. The following set of rules describes the syntax of a **<block>**:

```
<block>          ::= BEGIN <declaration> <statelist>
<statelist>      ::= ; <statelist>
<statelist>      ::= END
<statelist>      ::= <statement> <statelist>
```

In the Forth-79 implementation, we augment the meta-grammar with support structures for cyclic, optional and factored phrases however these are not supported in the Forth-83 implementation. The following example illustrates the use of these structures:

```
<block> ::=     BEGIN <statement> $( ; <statement> ) END
```

A block structure is defined as starting with **BEGIN**, followed by one or more statements separated by a semicolon, and finally **END**. The dollar sign **($)** modifies the phrase delimited by parenthesis so that it becomes a cyclic structure that repeats zero or more times. The specific details and grammar of the language definition will be discussed later.

Parsing Techniques

The meta compilers accept the LL(1) [AHO77] class of grammars. The Forth-79 implementation generates a predictive recursive descent parser [GRIE71] while the Forth-83 version implements an equivalent table driven parser.

The LL(1) class of parsers was chosen over the more powerful LR class primarily because they can incorporate semantic constructions within the production rule between any two components. LL(1) parsers chose the appropriate production rule based on the current state in the parse as well as the incoming token. Once made, the choice cannot be undone (i.e., no provision exists to backup and attempt another alternative). To allow backup would severely slow the parse down but more importantly, create difficulties in "undoing" any semantic actions that have been performed during the parse. Semantic actions represent the compile time actions of the compiler. These compile time actions include code generation for the program being compiled.

On the other hand, LR grammars [DERE71] do not choose the production rule until all of its components have been completely parsed. Holding off the production decision to this stage allows the parser to recognize a larger class of grammars and it allows LR grammars to be defined in a more flexible manner; for example it eliminates the need to factor productions and also permits left recursion. However, the semantic actions can only be executed at the end of any production. Stratification techniques which break down a large rule into smaller rules, are used to overcome this obstacle but result in a grammar that is less readable.

Further discussion of parsing techniques is beyond the scope of this article. The brief introduction presented here is simple intended to explain why we restricted the meta-compilers to the LL(1) class of grammars.

The Forth-79 Meta Compiler

The strategy of the Forth-79 meta compiler is to generate a word (or Forth definition) for each non-terminal symbol of the language description. This word is responsible for parsing the corresponding phrases and for execution of the requisite semantic actions.

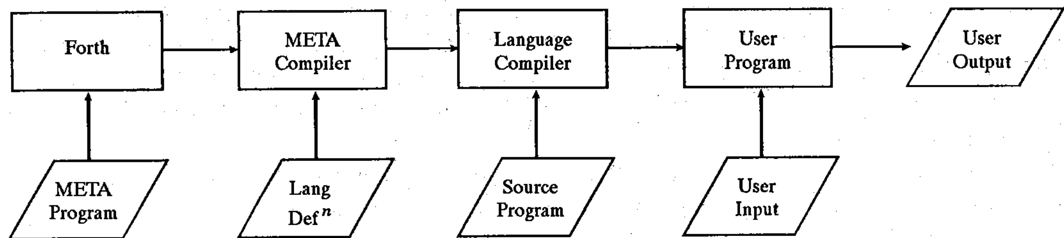

Figure 1. The META-79 version is a Forth program that, when loaded generates the META-79 compiler. Similarly, input of the language definition to the META-79 compiler generates the language compiler as another Forth program. Finally, input to the Language Compiler of a source program written in the defined language generates the User Program.

The interactions between the various elements are shown in Figure 1. The Forth source for the meta compiler is loaded into the Forth-79 system to generate the meta compiler. The BNF-like definitions are loaded by the meta compiler thus generating a language compiler as another Forth definition or function. Then, in order to create the user program, a source program written in the new language as defined by the previous definitions is parsed by the compiler. This compiler executes actions as defined by the semantics of the language definition. It creates the Forth function that implements the semantics of the user program.

An example of a language definition accepted by the META-79 compiler is shown in Figure 2. The meta-language is a modified Backus-Naur notation. The meta compiler generates a recursive descent compiler that constructs a new Forth word given a source program written in that language. The meta compiler accepts near LL(1) grammars but does not check for LL(1) conditions.

A language definition starts with the Forth word **LANGUAGE**. The name of the compiler is defined after the word **MAIN** which is the last production of the language definition. The example in Figure 2 creates a recursive descent compiler called SMALLALGOL; its production goal is **<STATE>** (a statement). The goal symbol represents the eventual goal of the parser; once satisfied, the parsing is considered complete.

Each production definition starts with the symbol **DEFINE** whose first element is the non-terminal being defined followed by its definition. The definition is composed of a list of alternative phrases separated by the symbol "**|**". Alternatives may be grouped within "**( )**" symbols. If the group list is preceded with a "**$**", the group is repetitive. Each phrase is composed of terminals (the reserved words of the language) and non-terminals.

The first production of Figure 2 defines a non-terminal **<VALUE>** which contains three alternatives starting with **<NUMBER>**, **<VARIABLE>**, and "-" respectively. The last alternative consists of a further list of alternatives that includes **<NUMBER>**, **<VARIABLE>** and **<EMPTY>**. The non-terminals, **<NUMBER>** and **<VARIABLE>** have been predefined by the meta compiler. In most compilers, these are handled by the lexical analyzer as it can perform this particular task more efficiently than the parser.

The **<EMPTY>** serves to make the entire list of alternatives optional as the parse is satisfied in all cases where an alternative is not selected.

The definition terminates with a "**;**" symbol. The compiler generator completes the current definition and includes that non-terminal symbol in its symbol list.

A terminal symbol is defined in the grammar by enclosing it in single quotes as illustrated in the definition for **<IF>** by its terminal symbols **IF**, **THEN**, **ENDIF** and **ELSE**. During the parse,

these terminal symbols are identified as string constants and string comparisons are performed with the lexical entities of the source code.

```
LANGUAGE
  DEFINE <VALUE>        <NUMBER> {[ #^ [LITERAL] ]}
                        | <VARIABLE> {[ ID^ , ]} { @ }
                        | '-' (   <NUMBER> {[ #^ MINUS [LITERAL] ]}
                                | <VARIABLE> {[ ID^ , ]} { @ MINUS }
                                | <EMPTY>
                              ) ;
  DEFINE <PRIMARY>      <VALUE> | '(' <TERM> ')' ;
  DEFINE <FACTOR>       <PRIMARY> $ (  '*' <PRIMARY> { * }
                        | '/' <PRIMARY> { / } ) ;
  DEFINE <TERM>         <FACTOR>  $ (  '+' <FACTOR>  { + }
                        | '-' <FACTOR>  { - } ) ;
  DEFINE <COND>         <TERM> (  '=' <TERM> { = }
                        | '#' <TERM> { = Ø= } ) ;
  DEFINE <ASSIGN>       <VARIABLE> {[ ID^ ]} ':=' <TERM> {[ , ]} { ! } ;
  DEFINE <IF>           'IF' <COND> 'THEN' { ØBRANCH } {[ HERE ]} { ABORT }
                        <STATE>
                        ( 'ENDIF' {[ HERE SWAP ! ]}
                          | 'ELSE' { BRANCH } {[ HERE ]} { ABORT }
                                  {[ SWAP HERE SWAP ! ]}
                            <STATE> {[ HERE SWAP ! ]}
                            'ENDIF'
                        ) ;
  DEFINE <STATE>   <ASSIGN> |  <IF>  | <BLOCK> ;
  DEFINE <BLOCK> 'BEGIN' <STATE> $ ( ';' <STATE> ) 'END' ;
  MAIN SMALLALGOL <STATE> ;
```

Figure 2. Definition for the META-79 Small-Algol Compiler. Terminals are enclosed in single quotes, non-terminals in angle brackets, a subphrase in parens, alternative phrases are separated by a vertical bar. Words enclosed by braces are runtime actions while ({[ ]}) enclose compile time actions.

The ability of a top-down parser to incorporate semantics into its syntax definitions [KNUT71], and the intuitive nature of top-down parsing, explains its popularity. Semantics in this META compiler are written in native Forth code and surrounded by braces "{ }". A number of META words have been defined to support semantic construction such as described in Table 1. There is nothing magical about these words; they are simply support definitions created along with the meta compiler. Anyone is free to create new definitions or use existing Forth compiler support functions such as **ØBRANCH** to support the language compiler under development.

The definition of value illustrates semantic use and how its close relationship to syntax can be exploited. The first alternative,

```
DEFINE <VALUE> <NUMBER>  {[ #^ [LITERAL] ]}
```

describes the semantic action when a numeric value is found in the source code. The first Forth word, #^ places the numeric value of **<NUMBER>** on to the stack. The second word, **[LITERAL]** encodes the value on the top-of-stack into the program code being generated.

Figure 3 gives an example of the main compiler word generated by the META compiler (i.e., the output of the second stage in Figure 1) for SMALLALGOL.

Symbol	Explanation
{[	Start the beginning of a compile time block of semantic code. This code will be executed when a parse is completed to this point in the sourse code. These actions are performed during the compilation of the source code.
]}	End the semantic code block started by {[. The semantic code blocks are NOT recursive.
{	Start a block of run time semantic code. The words within these definitions are stored directly into the user program currently being compiled and hence are only executed when the user program is executed.
}	End the block of run time semantic code.
ID^	Place the address of the last found symbol onto the stack. This function is typically used when variables are found in the user program; for example in the use of the <VALUE>. When <VARIABLE> is parsed, the semantic code will insert the <VARIABLE>s symbol table address onto the stack where the Forth word "," will insert it into the current user program definition.

Table 1. META-79 Semantics Construction Support Words.

The phrase **GETOKEN MAKE** creates a new word in the Forth dictionary and names the user program. The code for a statement is parsed by calling the word generated by META called **<STATE>**. On return, the status is checked and if **<STATE>** was successfully parsed, the semantic code between the **IF** ... **ENDIF** is executed. Finally the definition for the user program is completed with **CLOSEDEF**.

```
: SMALLALGOL GETOKEN MAKE
        <STATE> ?STATUS
        IF ENDIF
        CLOSEDEF
;
```

Figure 3. Forth Code Generated for SMALLALGOL.

An example of a non-terminal symbol is shown in Figure 4. From the **SMALLALGOL** definition, an **<IF>** phrase is

```
IF <COND> THEN <STATE> ENDIF
```

or

```
IF <COND> THEN <STATE> ELSE <STATE> ENDIF
```

The word definition generated by the META compiler for **<IF>** (see Figure 4) determines if the user program starts with **IF**. This recognition is performed through the word **SYMBOL** in a fashion similar to that of the standard Forth word **."** (dot quote). If the input source does not start with **IF**, a failure would be returned to the caller of this phrase, in this case, **<STATE>**; otherwise it attempts to parse **<COND>**. If the phrase fails here however, **?CHKERR** will print an error message and stop the entire parse process. It is not sufficient to report an error to the caller as there is no alternative to **<COND>** in the language definition of Figure 2 and therefore the input source must be wrong. In not finding an **IF** keyword, it is necessary to report back to **<STATE>** and let **<STATE>** determine what action to take as there may be an alternative to **<IF>**.

After a successful parse of **<COND>** and **<STATE>**, it expects either an **ENDIF** or **ELSE**. If neither is found it prints an error report and ends.

The code generated by **<IF>** for the program of Figure 5 is shown in Figure 6. As the sample input is parsed for **<IF>**, the compiler constructs code for the user program, **DEMONSTRATE**. Following the **<COND>**, the first action is to insert a **ØBRANCH** into the user program code (see

Figure 4) and save this location so that the branch target can be resolved later in the parse as the address cannot be determined until the first <STATE> has been constructed. Code following ENDIF resolves this address so that the branch target follows <STATE>. The ELSE semantics are more complicated in that the end of the first <STATE> contains an unconditional branch to the instruction that follows the <IF> phrase. The target of the first branch is set to be the location of the second <STATE>. The code generated by the program is shown in a stylized form by Figure 6. This branch example relies heavily on the already existing support structure for the conventional Forth branch words.

The Forth Meta-83 Compiler

A different system was developed under Forth-83 and Pascal. It was based on a table driven LL(1) grammar. Tables containing syntax and semantic descriptions generated by a Pascal program are fed to the Forth-83 based table driven syntax interpreter. The interpreter generates a Forth executable program when given a input in the language defined by the tables. Figure 7 shows the interrelationships between the various phases of system.

```
: <IF>
   SYMBOL IF                    ( Is the token an 'IF' Terminal?)
   DROP ?STATUS                 ( Drop the Code Field Address and check result)
   IF <COND> ?CHKERR            ( Yes, Parse for <COND> and check result )
     SYMBOL THEN                ( Is the token a 'THEN' )
     DROP ?CHKERR               ( Check result, report an error if required )
     INSERT ØBRANCH             ( Generate BRANCH if TOS=Ø code )
     HERE                       ( Store the BRANCH operand location )
     INSERT ABORT               ( Place an ABORT destination here temporarily )
     <STATE> ?CHKERR            ( Parse for <STATE> and report error )
     SYMBOL ENDIF               ( Is the token an 'ENDIF' )
     DROP ?STATUS               ( Drop CFA and check result )
     IF HERE SWAP !             ( Yes, resolve prev. branch operand )
     ELSE
       SYMBOL ELSE              ( No, check for and 'ELSE' )
       DROP ?STATUS             ( Is it an ELSE )
         IF  INSERT ØBRANCH     ( Yes, Insert another ØBRANCH )
           HERE INSERT ABORT    ( Save location and set ABORT target)
           SWAP HERE SWAP !     ( Resolve the first branch to here )
           <STATE> ?CHKERR      ( Parse <STATE> and report an error )
           HERE SWAP !          ( Resolve the second branch to here )
           SYMBOL ENDIF         ( Check for and 'ENDIF' )
           DROP ?CHKERR         ( And report and error if necessary)
         ENDIF                  ( No 'ELSE' )
         ?CHKERR                ( Report any errors )
       ENDIF
     ENDIF
;
```

Figure 4. Forth Code Generated by META for the Non-Terminal <IF>

```
SMALLALGOL DEMONSTRATE
BEGIN
  IF A = Ø THEN A := 1 ELSE A := Ø ENDIF
END .
```

Figure 5. User Program for the SMALL-ALGOL Language.

```
: DEMONSTRATE
        A @ Ø = IF
                  1 A !
                ELSE
                  Ø A !
                ENDIF
;
```

Figure 6. Stylized Forth Code Result of Compiling the User Program.

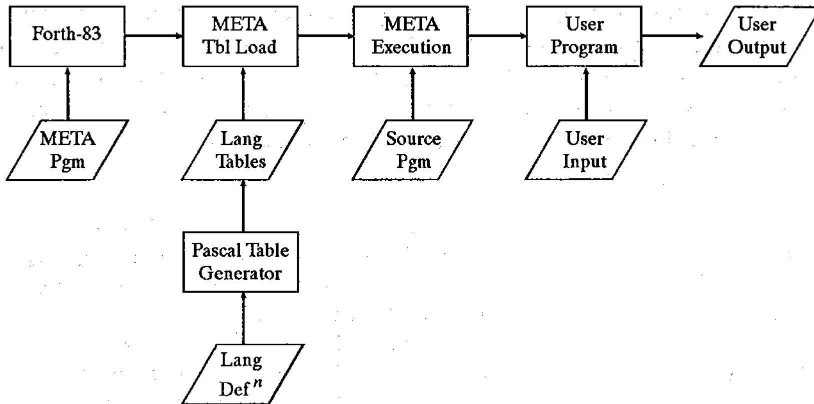

Figure 7. META-83 Compiler Flow for a Table Driven Compiler.

The language description is fed to a Pascal based table generator that generates two language tables: the parse syntax/semantics and (non)terminal symbols of the target language. After the code for the META-83 system is loaded into the Forth engine, the META-83 system is instructed to load the tables and initialize the various data structures required for compilation.

Unlike the META-79 system, META-83 does not create new Forth definitions that together form the new target compiler. Instead the target compiler is simply the META-83 system combined with the language unique tables already generated. The META-83 system can be used for as many different languages as tables exist. It typically generates as part of its semantic structure, the user program as a conventional Forth word that is executed as if the word where defined in Forth source code.

An example language definition for a more complete Algol is given in Figure 8. This language description is in a strict Backus-Naur Form. It must conform to LL(1) grammar restrictions; it must be factorized, devoid of left recursion and the director symbol sets for each phrase must be disjoint. The director symbol set represents those terminal symbols that can start a production rule (or definition).

The language format consists of a series of production definitions separated by semicolons. The language description ends with a period. A non-terminal may be defined by multiple rules each representing an alternative. The definition may be empty but there cannot be any direct or indirect left recursion within the grammar. A left recursion is a situation where the first item of a production rule is the non-terminal being defined (direct recursion) or through other rules can be the nonterminal being defined (indirect recursion). To do so would violate the precondition that the director sets of all alternatives be disjoint and could lead to a state table with circular definitions.

The two special terminal symbols, **IDENT** and **NUMBER** are predefined. As in the META-79 system, these symbols represent the lexical entities **<IDENTIFIER>** and **<NUMBER>** respectively.

```
<PROGRAM>       =   IDENT { TOKEN DEF -1 BLOCK ! -1 LEXLEVEL ! }
                    <BLOCK> . { COMPILE UNNEST }                    ;
<ID>            =   IDENT { LEXLEVEL @ TOKEN SEARCHIDLEX DUP
                            IF DUP NIL =
                              IF FATAL" ID NOT IN SCOPE" THEN
                            ELSE FATAL" UNDECLARED ID" THEN }       ;
<VALUE>         =   <ID> { [COMPILE] LITERAL [ @VAR ] }             ;
<VALUE>         =   NUMBER { TOKENVAL @ [COMPILE] LITERAL }         ;
<PRIMARY>       =   <VALUE>                                         ;
<PRIMARY>       =   ( <TERM> )                                      ;
<PRIMTAIL>      =   * <PRIMARY> { [ * ] }<PRIMTAIL>                 ;
<PRIMTAIL>      =   / <PRIMARY> { [ / ] }<PRIMTAIL>                 ;
<PRIMTAIL>      =                                                   ;
<FACTOR>        =   <PRIMARY> <PRIMTAIL>                            ;
<FACTAIL>       =                                                   ;
<FACTAIL>       =   + <FACTOR>  { [ + ] } <FACTAIL>                 ;
<FACTAIL>       =   - <FACTOR>  { [ - ] } <FACTAIL>                 ;
<TERM>          =   <FACTOR> <FACTAIL>                              ;
<DECLARATION>   =   IDENTIFIER <DECLIST>                            ;
<DECLARATION>   =                                                   ;
<DECLIST>       =   IDENT { TOKEN ENTERID } <NEXTDEC>               ;
<NEXTDEC>       =   , <DECLIST>                                     ;
<NEXTDEC>       =   ';                                              ;
<BLOCK>         =   BEGIN { 1 LEXLEVEL +! 1 BLOCK +! } <DECLARATION>
                          <STATELIST>                               ;
<STATELIST>     =   '; <STATELIST>                                  ;
<STATELIST>     =   END  { LEXLEVEL @ 1- LEXLEVEL ! }               ;
<STATELIST>     =   <STATEMENT> <STATELIST>                         ;
<STATEMENT>     =   <BLOCK>                                         ;
<STATEMENT>     =   <ASSIGN>                                        ;
<STATEMENT>     =   <IF>                                            ;
<STATEMENT>     =   EXIT                                            ;
<STATEMENT>     =   OUTPUT <TERM> { [ CR . ] }                      ;
<STATEMENT>     =   FOR <ID> =
                        <TERM> { DUP [COMPILE] LITERAL [ !VAR ] }
                        TO { HERE }
                        <TERM> { SWAP DUP [COMPILE] LITERAL
                                 [ @VAR >= ?BRANCH ]  HERE Ø , }
                        DO
                        <STATEMENT> { SWAP DUP [COMPILE] LITERAL
                                      [ @VAR 1 + ] [COMPILE] LITERAL
                                      [ !VAR BRANCH ] SWAP , HERE SWAP ! };
<ASSIGN>        =   <ID> = <TERM> { [COMPILE] LITERAL [ !VAR ] };
<IF>            =   IF <COND> THEN
                     { COMPILE ?BRANCH HERE Ø , } <STATEMENT> <ELSEPART>;
<ELSEPART>      =   ELSE { COMPILE BRANCH HERE Ø , SWAP HERE SWAP ! }
                     <STATEMENT> { HERE SWAP ! }                    ;
<ELSEPART>      =   { HERE SWAP ! }                                 ;
<COND>          =   <TERM> '= <TERM> { [ = ] }                      ; .
```

Figure 8. Language definition used to generate the tables used by the META-83 compiler.

Meta symbols such as "=" can be part of the language when they are preceded by a single quote. The semantic descriptions containing the Forth-83 words required to implement the language, are enclosed in braces, ("{" and "}") and are executed when the structure associated with it is parsed. Within the semantic description, code within brackets ("[" and "]") is inserted into the program being generated.

Table Formats

The META-83 parser uses two tables generated by a Pascal program. The first table contains the keywords or reserved words of the language to direct the lexical analyzer. The analyzer built into the meta system returns unique values for each of these tokens when they are found in the input as defined by the table.

An `<IDENTIFIER>` is predefined by the lexical analyzer, however its semantic actions are the responsibility of the semantic constructions within the language definition. It is possible for one language to interpret identifiers within a FORTRAN-like flat scope structure, or a ALGOL-like static block structure, or a LISP-like dynamic structure. Thus, the semantic actions must define symbol table structure and construction, search techniques, access techniques, data-storage mechanisms and so on. Of course, these semantic actions could be predefined Forth definitions that are simply called upon by the semantic analyzer at compiler time.

Parsing Action Table

A parsing action table is read by the META-83 system in order to direct the syntax and semantic actions. It is composed of mode lines followed by semantic actions each consisting of a mode, production number, its action fields and a director list.

The mode field describes the contents of the other fields. There are four modes:

- "3". End of Parsing Table.
- "2". Definition for non-terminal whose index entry is in *field*$_2$. A third field, *field*$_3$ is a flag that defines error actions, e.g., **2 Ø Ø**. This defines non-terminal referenced as "0" with no error actions.
- "1". Defines the beginning of a production rule for the non-terminal and the actions for the first grammar symbol of this alternative. It contains the three action entries for the grammar symbol (either terminal or nonterminal) of *field*$_2$ in *field*$_3$ through *field*$_5$. These fields are followed by a list of director symbols. The special value 255 signifies the end of the director list. By definition, the director list references only terminal symbols. A non-terminal may have many definitions representing alternative production rules. The director list and the current input symbol in the source language select the appropriate production rule during a parse. The rule may also contain semantic actions that are to be taken whenever the current grammar symbol is recognized and parsed.
- "0". Defines the action for the next grammar symbol (*field*$_2$) within the alternative production. This contains the same information as the previous mode selector. This item is place in order of occurrence within the production definition and is essentially the same as mode 1 but does not begin a new production rule.

The entry *field*$_2$ in modes 0 and 1 can represent either a unique terminal or non-terminal grammatical symbol. The interpretation depends on the action fields. For instance the action **STACK** or **JUMP** can only apply to non-terminals. The action **ACCEPT** applies only to terminal symbols.

Consider the production rules for **<PRIMTAIL>**. Its unique non-terminal representation is 6 as seen in Figure 9. The relevant production rules are:

```
<PRIMTAIL> ::= *<PRIMARY> <PRIMTAIL>
<PRIMTAIL> ::= /<PRIMARY> <PRIMTAIL>
<PRIMTAIL> ::=
```

An example parse table is illustrated in Table 2.

Mode	Symbol	Action	Directors	Semantics	Code
Define	<PRIMTAIL>	No Error			2 6 Ø
Alternative	*	Accept	*		1 5 255 Ø Ø 5 255
Action	<PRIMAR>	CALL		[*] ;SEM	Ø 4 Ø 255 255
Action	<PRIMTAIL>	JUMP			Ø 6 Ø 255 Ø
Alternative	/	Accept	/		1 6 255 Ø Ø 6 255
Action	<PRIMARY>	CALL		[/] ;SEM	Ø 4 Ø 255 255
Action	<PRIMTAIL>	JUMP			Ø 6 Ø 255 Ø
Alternative	ϵ	Accept	) + - ; END = TO DO THEN ELSE		1 23Ø Ø Ø Ø 4 7 8 11 13 17 18 19 21 22 255

Table 2. A portion of the parse table for the language defined in figure 8.

The three action fields represent the functions **ACCEPT**, **JUMP** and **STACK** respectively. An **ACCEPT** action applies to terminal symbols. The parser will **ACCEPT** the current input symbol if it is the same as the given symbol in $field_2$. On a successful match, the next input token is read in. Otherwise an error message is printed and it may attempt to synchronize to a synchronization symbol if the table defines the error action code. The **JUMP** and **STACK** action only apply to non-terminal symbols. If a phrase needs to be parsed, the **JUMP** field is set and when called as a subroutine, the **STACK** field is also set. This supports an efficient implementation of tail recursive parsing. The last non-terminal symbol of a production rule does not have to stack its return parse location so long as semantics do not follow the action.

TERMINALS	
IDENT	Ø
NUMBER	1
.	2
(	3
)	4
*	5
/	6
+	7
-	8
IDENTIFIER	9
,	1Ø
;	11
BEGIN	12
END	13
EXIT	14
OUTPUT	15
FOR	16
=	17
TO	18
DO	19
IF	2Ø
THEN	21
ELSE	22

Non-TERMINALS	
PROGRAM	Ø
BLOCK	1
ID	2
VALUE	3
PRIMARY	4
TERM	5
PRIMTAIL	6
FACTOR	7
FACTAIL	8
DECLARATION	9
DECLIST	1Ø
NEXTDEC	11
STATELIST	12
STATEMENT	13
ASSIGN	14
IF	15
COND	16
ELSEPART	17

Figure 9. Parser Values for Terminal and Non-Terminal Grammatical Symbols.

A list of director symbols is given for each alternative entry. They is used by the parser to identify the proper production rule given the current input token. If none of the alternatives contain a director symbol that matches the current token, an error is flagged.

An empty production rule has a special value, in this case 230, encoded into $field_2$. If any of the director symbols for the rule is found as an input token, they will be accepted, but the input token is retained.

Semantics

Semantics are defined in the native code of the parser support system, here, Forth-83. However, the code is not in the form of a Forth definition but rather as a segment of a Forth definition. Whenever a symbol has been correctly parsed, the semantic actions that follow the production symbol are executed. These actions may be symbol table management, code generation of the user program, code management (such as forward address resolution) and so on.

An example is shown for the `FOR` statement of Figure 8. Given the source in Figure 10, having parsed the phrase `<ID>`, the semantic code will place the identifier's address onto the semantic stack after searching the symbol table. The phrase `<TERM>` will generate code that evaluates a term into the user program. The identifier address is duplicated and placed into the user program as a `LITERAL` along with `!VAR`. The address duplication is necessary for future use of `<ID>`. To this point we have generated the following code:

```
1 A !VAR
```

Following the second `<TERM>` and its code generation, the address of the identifier is compiled into the code along with `@VAR >= ?BRANCH Ø`. The `Ø` is placed as a temporary position holder for the `ØBRANCH` target address. A `HERE` instruction is used to save this location for later resolution.

Finally, after `<STATEMENT>` and its code is generated, the final identifier increment code

```
A @VAR1 + A !VAR
```

is generated. Another branch instruction is inserted so that the process will branch to the start of the loop retained by the initial `HERE` instruction. The first branch address is resolved so that its target follows the `FOR` loop.

The Block Environment

As previously alluded to, the mechanism of identifier storage is left up to the discretion of the language designer. It is not necessary to use the standard Forth dictionary scheme; instead it

```
GENERATE TEST
BEGIN
  IDENTIFIER A;
  FOR A = 1 TO 3+4 DO OUTPUT A;
  OUTPUT A
END
SEE TEST
: TEST   (LIT) 1 (LIT) 28998 !VAR                ( Initialize A )
    (LIT) 3 (LIT) 4 +                            ( Calculate limit )
    (LIT) 28998 @VAR >=                          ( A >= limit? )
    ?BRANCH 32                                   ( Yes, branch over code )
      (LIT) 28998 @VAR CR .                      ( No, print value )
      (LIT) 28998 @VAR 1 + (LIT) 28998 !VAR      ( Increment A )
      BRANCH -5Ø                                 ( Branch to limit check )
    (LIT) 28998 @VAR CR . ;                      ( Print final value of A )
```

Figure 10. Source Program and User Code Produced by the META-83 Compiler.

is possible to design other access mechanisms. In the META-79 version, identifier access was merely an extension of the Forth dictionary and no measures were take to create new access mechanisms. Here we define a new structure that implements a more sophisticated form for the purposes of illustration.

The identifier "A" is not in the program as a Forth word but instead it is an address of a descriptor block. In this illustration, a simple nested environment structure has been constructed in Forth to closely mimic that of ALGOL. This set of definitions includes symbol table maintenance, variable access, and environment maintenance.

This environment generates descriptors for each occurrence of a variable in the program along with a list of values that it can take (for the purposes of recursion). Separate variables have a common symbol table entry that points to this list of descriptors. The general format is shown in Figure 11.

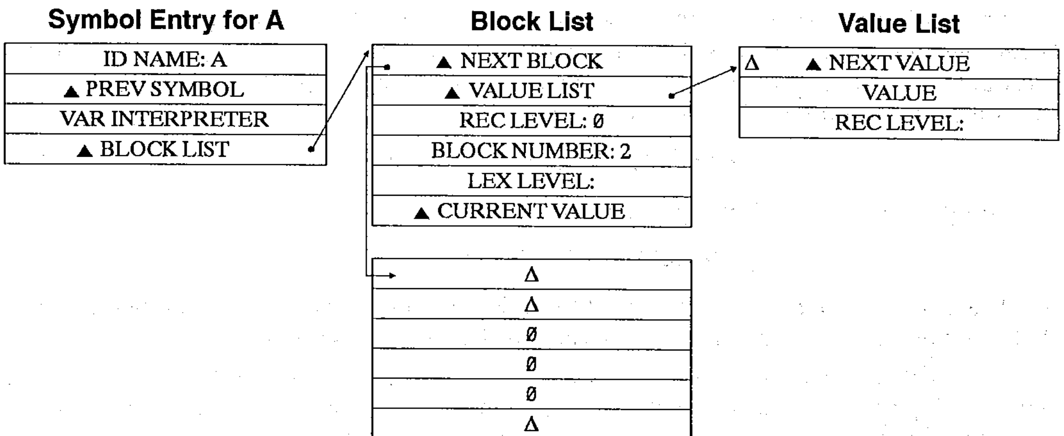

Figure 11. META-83 ALGOL Environment Block structure.

The compiler, when given a variable, will search the symbol table for the given name and return the correct descriptor based on a lexical analysis. The example in Figure 12 generates the block structure of Figure 11. The position pointed to by the arrow would allocate a value block at run time and insert itself into the value list for block 2, lexical level 1. If it had not been found at the current lexical level, the search would have continued at the next lower level. If *A* were referenced in the second block (lexical level 1), the search would find *A* in the parent block at lexical level 0. This block address would be inserted into the code generated for this program.

Discussion

These meta compiler systems offer significant advantages over hand coded techniques of Forth compiler extension. They automate the construction of the parser system, leaving the programmer to define the syntax of the language in a well known notation (BNF) and its semantics. Forth provides a strong environment to implement these semantics. In particular, the Forth compiler functions are available for the semantic descriptions and the stack provides a simple mechanism to implement nested structures such as expressions and declarations of block structured languages.

```
BEGIN
    IDENTI A;
    BEGIN

    END
    BEGIN
        IDENTI A;
        A := 1;                  <-

    END
END
```

Figure 12. META-83 ALGOL Language Block Environment. This structure permits resolution of a unique object within the scope environment.

Possible Applications and Extensions

The Forth-79 implementation provides a very simple way to implement extensions to the Forth compiler. For example, we have defined a Forth immediate word, "{" (a Forth word, not a lexical entity), whose purpose is to implement an infix arithmetic phrase compiler that can be used within a conventional Forth definition, e.g.,

```
: DEMONSTRATE { A + B / C } . ;
```

This generates a definition equivalent to:

```
: DEMONSTRATE A @ B @ C @ / + . ;
```

This could be extended to construct ALGOL's `FOR` statements and so on using the simple BNF like notation and semantic structures provided by the meta-compiler.

The techniques described here are not limited to compiler applications. It is possible to generate a high-level language to machine code compiler (as opposed to the Forth engine code). Instead of the semantics generating Forth code structures, it could produce binary structures or, in a manner similar to the UNIX PCC compiler, assembler code. The system could also act as a pre-processor, similar to that to the UNIX CPP preprocessor system. The pre-processed code would then be processed by say, the Forth compiler. Simple front end application languages are easily implemented using this mechanism. For instance, a data base application whose primitive functions are written in conventional Forth code could implement the user input interpreter with the meta compiler. The language would describe the acceptable forms of data base commands and its semantics could describe the appropriate functions that would be invoked for these commands.

The META-83 implementation does not create new Forth functions that act as compilers, but accepts tables with a common compiler kernel instead. This system is more flexible as many tables can reside within the Forth vocabulary and improvements can be made to the table generator independent of the kernel system.

A problem arose in the construction of the META-83 implementation. Specifically, the vocabulary structure based on the stack is not a suitable environment for the simultaneous generation of definitions. In our case, an allocation of vocabulary space was set aside for the symbol table entries that were generated while the program was also being generated. If the allocation proves inadequate, the only recourse is to abort the compilation. The META-79 implementation did not handle variable or constant declarations but left that to the programmer to create beforehand. Of course it could have been extended to include such capabilities and the same problem would have arisen.

Lexical Analysis

The lexical analysis used by Forth using the function **WORD** delimits the lexical entities of the language with "white-space." Many languages use separators other than spaces or new-lines. For instance, the ALGOL system uses special characters to delimit the keywords or operators in the language. To complete this system, a new function similar to **WORD** should be constructed. Ideally this system would accept a variety of different analyzers similar to that of LEX **[SCHR85] [JOHN78]**.

Pascal?

The Meta-83 implementation relied on data structures generated by an external program written in Pascal. We did this as we felt that Pascal was better suited than Forth to the generation of our data structures. Forth provides an extremely extensible environment; Pascal supports complex data structures that lend themselves well to the purposes of recursive structure analysis using lists and non-homogeneous data sets. While this could have been duplicated in Forth, we did not feel it necessary to pursue this option. However, having completed the first phase of the metacompiler, it seems that an appropriate test of its power would be to implement some of the more useful Pascal facilities into the basic Forth engine and then perform a port of the table generation code to this system.

MLL

This work was targeted toward the implementation of a new engine MLL (Multiple Level Language) whose internal structure was based upon list structures rather than stacks. Yet it implemented the simplicity of the internal Forth engine (i.e., threaded code with multiple interpreters). It was our intention to learn from our experience with Forth and the meta-compiler in order to generate a new language system where a number of interpreters were defined and operating upon a single data structure (the list) just as Forth operates on stacks for both its data structures and word definitions. We have been successful in constructing a simple prototype of this engine.

Source Code

The source code for the META-79 implementation is given in Appendix A. The META-83 implementation (including the Pascal program) can be obtained by writing to the authors at the above address or via email to:

"FS300022@sol.yorku.ca"

or

"tyler@stpl.ists.ca".

References

[AHO77] Alfred V. Aho and Jeffrey D. Ullman, *Principles of Compiler Design*, Addison-Wesley Series in Computer Science and Information Processing, Addison-Wesley Publishing Co., Reading, Massachusetts, 1977.

[DERE71] Franklin L. DeRemer, "Simple LR(k) Grammars", *Communications of the ACM*, 14(7):453-460, July 1971.

[DERI82] Mitch Derick and Linda Baker, *Forth ENCYCLOPEDIA*, Mountain View Press, Mountain View, CA, 2nd edition, 1982.

[F-83] Forth Standards Team, *Forth-83 STANDARD*, Mountain View Press, Mountain View, CA, August 1983.

[GRIE71] David Gries, *Compiler Construction for Digital Computers*, John Wiley & Sons, Inc., New York, 1971.

[HUNT81] Robin Hunter, *The Design and Construction of Compilers*, Wiley Series in Computing, John Wiley & Sons, Inc., Chichester, 1981.

[JOHN78] Stephen C. Johnson, *YACC: Yet Another Compiler-Compiler*, Technical Report, Bell Laboratories, Murray Hill, New Jersey, July 1978.

[KNUT65] Donald E. Knuth, "On the Translation of Languages from Left to Right," *Information and Control*, **8**:607-639, 1965.

[KNUT71] Donald E. Knuth, "Top-Down Syntax Analysis," *Acta Informatica*, **1**:79-110, 1971.

[KORE69] A. J. Korenjak, "A Practical Method for Constructing LR(k) Processors," *Communications of the ACM*, **12**(11), November 1969.

[PYST80] Authur B. Pyster, *Compiler Design and Construction*, Electrical/Computer Science and Engineering Series, Van Nostrand Reinhold Co., New York, 1980.

[SCHR85] Axel T. Schreiner and H. George Friedman, Jr., *Introduction to Compiler Construction with UNIX*, Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1985.

Dr. Tyler Ivanco received his B.Sc. in Chemistry from the University of Lethbridge in Alberta, Canada in 1979 and his Ph.D. in Theoretical Chemistry from York University, Toronto, Canada in 1987. Currently he is with ITRES Research in Calgary, Alberta. For the past ten years he has worked on hardware and software used in interfacing and networking. He is experienced in all aspects of PC board design and fabrication as well as system and application programming in languages including assembler, Pascal, C and LISP.

Dr. Geoffrey Hunter received his Grad. R.I.C. in Chemistry from the University of Salford, England in 1961 and his M.Sc. (1962) and Ph.D. (1965) in Theoretical Chemistry at Manchester University, England. He has been on the Faculty of the Department of Chemistry at York University since 1966. His research interests include: quantum chemistry, computer architecture and language design, and the fundamentals of quantum physics and relativity theory. Dr. Hunter has taught Forth to Computer Science majors in a sophomore course about Machine Structures.

Forth was chosen as the implementation basis for Ivanco's Ph.D. research into the essential computational requirements for quantum chemistry, because it facilitated the definition of an extensible high-level language with its semantics defined via Forth words.

Appendix A. Forth-79 Meta Compiler Implementation

```
( Define a type SYMTABLE )
( Builds a SNUM sized table.  The table is referenced with an )
( index.  If it is -1, it adds to the next available location )
( otherwise it obtains the address of the given entry         )
: SYMTABLE       <BUILDS
( S E )                  DUP +  34 + DUP ,        ( SET B/SYM     )
( S B/SYM )              * Ø DO Ø C, LOOP         ( Ø TABLE       )
( S E A )        DOES>   SWAP >R                  ( SAVE ENTRY    )
( S A )                  DUP @ SWAP 2 +           ( GET B/SYM     )
( S B/SYM A' )           ROT 1 -                  ( ADJUST ROW    )
( B/SYM A' S-1 )         ROT * +                  ( INDEX ROW     )
( A'' )                  R> DUP Ø<                ( INDEX COLUMN  )
( A'' E E=Ø? )           IF DROP                  ( RETURN NAME A)
                         ELSE
( A'' E )                   DUP IF 1- DUP + 34 + + ( RETURN ENT )
( A''+{E-1}*2+34 )              ELSE DROP 32 +    ( RETURN #ENT   )
                                ENDIF
                         ENDIF ;
: ARRAY <BUILDS ALLOT DOES> ;
Ø                VARIABLE VAL            ( Current token value )
Ø                VARIABLE OLDVAL         ( Old token value )
32               ARRAY    TOKEN          ( Current token string )
32               ARRAY    OLDTOKEN       ( Old token string )
Ø                VARIABLE UNT'           ( Unit for a forward ref. )
Ø                VARIABLE SWITCH
5Ø               CONSTANT SNUM           ( Number of entrys )
15               CONSTANT SENTRY         ( Size of an entry )
SNUM SENTRY      SYMTABLE FTABLE         ( Create the table )
' USE CFA @      CONSTANT DOVAR          ( Save address of key interpreters )
' : CFA @        CONSTANT DOCOL
' ABORT CFA      CONSTANT DOABORT
: ENTERNAME ( NAMEADD SNUMBER - )  -1 FTABLE 32 CMOVE ;
: ENTERVAL  ( VALUE SNUMBER - )
                 DUP Ø FTABLE DUP DUP   ( GET # ENTRIES )
                 @ 1+ SWAP ! @
                 FTABLE ! ;             ( PLACE THE NEW VALUE  )
: GETNAME   ( NAMEADD SNUMBER - )  -1 FTABLE SWAP 32 CMOVE ;
: GETVAL    ( SNUMBER SENT - VALUE ) FTABLE @ ;
: GETENTRIES ( SNUMBER - NUM ENTRIES ) Ø FTABLE @ ;
```

```
Ø VARIABLE FLAG
( String Compare )
: COMPARE ( A B — ? | ?:=Ø UNSUCCESSFUL; =1 SUCCESSFUL        )
( A B )         1 FLAG !                ( SET COMPARE FLAG TRUE)
( A B )         DUP C@ 1+ Ø             ( SET DO LIMITS        )
( A B H Ø )     DO
( A B )           OVER OVER             ( COPY TOS, TOS-1      )
( A B A B )       I + C@                ( GET BYTE 1           )
( A B A B' )      SWAP I + C@           ( GET BYTE 2           )
( A B B' A' )     = FLAG @ AND FLAG !   ( FLAG:=B'=A'=FLAG     )
( A B )         LOOP
( A B )         DROP DROP FLAG @        ( DROP ADDRESSES AND   )
( ? )           ;                       ( RETURN COMPARE FLAG  )
( SEARCH FOR NAME IN FORWARD TABLE  )
: SEARCH  ( N—I | I:<Ø ABS{I}= EMPTY ENTRY;=Ø EOT;>Ø FOUND @ I)
( N )          1                   ( START AT ENTRY 1 IN TABLE)
( N I )        BEGIN
( N I )          OVER OVER -1 FTABLE ( ADDRESS ITH SYM NAME     )
( N I N S )      COMPARE             ( ARE THE TWO NAMES EQIV   )
( N I ? )        IF 1                ( YES..EXIT LOOP           )
( N I )          ELSE DUP SNUM @ =   ( IS THIS THE LAST ENTRY?  )
( N I I=SNUM )     IF DROP Ø 1       ( YES..RETURN EOT FLAG     )
( N I )            ELSE DUP -1 FTABLE C@ ( IS THIS AN EMPTY NAME)
( N I ? )            IF 1+ Ø         ( NO..INCR INDEX AND LOOP  )
( N I )              ELSE MINUS 1    ( YES..RETURN -I AS FREE   )
( N I L? )           ENDIF
( N I L? )         ENDIF
( N I L? )       ENDIF
( N I L? )     UNTIL SWAP DROP ; ( FORGET THE NAME ADDRESS )
( PLACE SYMBOL & VALUE INTO TABLE    )
: FORWARD         ( VALUE — )
( V )             OLDTOKEN SEARCH          ( LOOK UP TOKEN NAME    )
( V I )           -DUP
( V I )           IF DUP Ø<                ( IF PARTIAL SUCCESS    )
( V I I<Ø )         IF MINUS DUP           ( IF NOT FOUND I=FREE   )
( V I I )             OLDTOKEN SWAP ENTERNAME   ( STORE NAME       )
( V I )             ENDIF
( V I )             ENTERVAL               ( AND ALWAYS STORE VAL  )
                  ELSE DROP ." FORWARD TABLE FULL" CR QUIT
                  ENDIF ;
```

```
( RESOLVE the symbol table value )
: RESOLVE       ( - )
                SNUM 1+ 1               ( FOR ALL NAMES          )
( H 1 )         DO I GETENTRIES -DUP    ( GET ENTRY/NAME         )
( E E )            IF TOKEN I GETNAME   ( COPY NAME INTO TOKEN )
( E )                 TOKEN CONTEXT @ @ (FIND) ( & DO A LOOKUP )
( E PFA L ? )         IF DROP CFA       ( IT'S FOUND, GET CFA  )
( E CFA )             ELSE DOABORT
( E CFA )                TOKEN COUNT TYPE ." NOT FOUND" CR
( E CFA )             ENDIF
( E CFA )             I ROT 1+ 1        ( RESOLVE EACH ENTRY   )
( CFA I E+1 1 )       DO OVER OVER I GETVAL ! ( RESOLVE ADDRESS)
( CFA I )             LOOP DROP DROP    ( CLEAR UP THE STACK   )
( )                ELSE LEAVE           ( NO MORE ENTRIES      )
                   ENDIF
                LOOP ;
: ?CHKERR       SWITCH @ Ø= IF ." ERROR IN SYNTAX AT: "
                               TOKEN COUNT TYPE CR QUIT ENDIF ;
: SELF          LATEST  PFA     CFA     , ; IMMEDIATE ( Recursion )
: INSERT        R> DUP 2 + >R @ , ; ( Insert the next symbol in defn )
: SET           1 SWITCH ! ;        ( No error )
: RESET         Ø SWITCH ! ;        ( Error )
: NEXTBLK       1 BLK +! Ø IN ! ;   ( Read in the next block )
( Get the next token from the input stream )
: GETOKEN       TOKEN   OLDTOKEN        32      CMOVE
                BEGIN   32      WORD    HERE    1+      C@
                        IF      Ø
                        ELSE    BLK     @
                                IF      NEXTBLK
                                ELSE    QUERY CR
                                ENDIF   1
                        ENDIF
                WHILE
                REPEAT  HERE    TOKEN   32      CMOVE   ;
( Is it a digit? )
: NUM           DUP     47      >       SWAP    58      <
                AND     ;
( Is it a letter? )
: LETTER        95      AND     DUP     64      >       SWAP
                91      <       AND     ;
( Is it a letter or digit? )
: LETNUM        DUP     LETTER  SWAP    NUM     OR          ;
: SETSWITCH     DUP     SWITCH  !
                IF      GETOKEN ENDIF   ;
: ?STATUS       SWITCH  @       Ø=      ;
( Insert a branch on true condition )
: BRANCHTRUE    INSERT  ?STATUS INSERT  ØBRANCH ;
( Insert a branch on false condition )
: BRANCHFALSE   INSERT  ?STATUS INSERT  Ø= INSERT  ØBRANCH ;
```

```
( Predefined non-term <VARIABLE>, looks a variable up in the vocabulary )
: <VARIABLE>    TOKEN   DUP     1+      C@      LETTER
                IF      1       SWAP    COUNT   Ø
                        DO      DUP     I       +       C@
                                LETNUM  ROT     AND     SWAP
                        LOOP    DROP
                        IF TOKEN CONTEXT @ @ (FIND)
                           IF DROP CFA @
                              DOVAR =
                           ELSE TOKEN COUNT TYPE ." NOT FOUND" Ø
                           ENDIF
                        ELSE Ø
                        ENDIF
                ELSE    DROP Ø
                ENDIF   SETSWITCH ;
( Lookup a non-terminal )
: NONTERMINAL   TOKEN   1+      C@      6Ø      =
                IF      TOKEN   COUNT   1       -       +       C@
                        62      =
                        IF      TOKEN   CONTEXT @  @  (FIND)
                                IF DROP CFA 1
                                ELSE Ø -1
                                ENDIF
                        ELSE    Ø
                        ENDIF
                ELSE    Ø
                ENDIF   DUP     IF      GETOKEN ENDIF ;
( Construct a <NUMBER> )
: <NUMBER>      Ø.      TOKEN
                (NUMBER)        C@      BL      -
                IF      DROP    DROP    Ø
                ELSE    DROP    VAL     DUP     @ OLDVAL ! !    1
                ENDIF
                SETSWITCH       ;
( An empty production )
: <EMPTY>       SET ;
( Print an error message )
: ?ERR          SWAP
                IF      DROP    1
                ELSE    CR      ." ILLEGAL SYNTAX:"  .
                        ." FOUND AT:" TOKEN COUNT TYPE
                        QUIT
                ENDIF   ;
```

```
( Support function that matches an input token with the )
( String that follows this entry in the program definition )
: (SYMBOL)
                TOKEN   R       COUNT   +       R>      SWAP
                >R      DUP     C@      1+      1       SWAP    Ø
                DO      ROT     ROT
                        OVER    OVER
                        I       +       C@      SWAP
                        I       +       C@      =
                        >R      ROT     R>      AND
                LOOP
                ROT     ROT     DROP    DROP
                DUP     SETSWITCH       ;
: SYMBOL
        COMPILE (SYMBOL)
        32      WORD    HERE    C@      1+      ALLOT
;       IMMEDIATE
( Obtain a string )
: STRING
        TOKEN   1+       C@     39      =
        IF      TOKEN   COUNT   1       -       +       C@
                39      =       IF      1
                                ELSE    Ø
                                ENDIF
        ELSE    Ø
        ENDIF
        DUP     IF      GETOKEN ENDIF ;
( Predefined non-terminal for <STRING> )
: <STRING>      STRING  DUP     SWITCH  !  ;
: CPYSTG        OLDTOKEN        C@      2       -       C,
                OLDTOKEN        COUNT   1 -     DUP     1 -
                IF      1
                        DO      DUP  I   +  C@  C,
                        LOOP
                        DROP
                ENDIF   ;
: TSTSTG        INSERT  (SYMBOL)
                CPYSTG
                INSERT  DROP    ;
: } ;
( Build semantic code into the current non-terminal definition )
: {     ( - )  ( INSERTS CODE WORDS BETWEEN { AND } INTO WORD )
                R>                      ( OBTAIN 1ST ADD PAST })
( A )           BEGIN DUP DUP
( A A A )         @ [ ' } CFA ] LITERAL =
( A A CFA=} )    Ø= WHILE               ( WHILE NOT } DO        )
( A A )           @  ,  2 +             ( INSERT CODE INTO WORD)
( A )           REPEAT                  ( AND MOVE PAST IT      )
( A A )         DROP 2 + >R  ;          ( RESTORE PAST THE }    )
( Insert a literal constant )
: [LITERAL]     INSERT  LIT , ;
```

```
( Insert a double sized literal )
: [DLITERAL]    SWAP [LITERAL] [LITERAL] ;
( Insert a function or number into the definition )
: TRANSLATE  ( - ?:=Ø NUMBER; <> Ø CFA  )
      TOKEN CONTEXT @ @ (FIND)
      IF      DROP    CFA     DUP    ,
      ELSE    TOKEN   NUMBER  DPL    @ 1+
              IF      [DLITERAL]
              ELSE    DROP [LITERAL]
              ENDIF
              Ø
      ENDIF   ;
: ]} ;
( Place the token value onto the stack )
: #^          VAL @  ;
( Place the code field address onto the stack )
: ID^         OLDTOKEN CONTEXT @ @ (FIND)
              IF DROP CFA
              ELSE ." IDENTIFIER:" OLDTOKEN COUNT TYPE
                   ." NOT DEFINED" CR
                   DOABORT
              ENDIF   ;
( Store the definition into the compiler for execution at compile time )
: PERFORM          ( - ? | :=Ø NUMBER :<>Ø WORD )
              BEGIN
                TRANSLATE  DUP
                IF    [ ' ]} CFA ] LITERAL =
                ENDIF
                GETOKEN
              UNTIL
              1          ;
( Store the semantics into the compiler for inclusion into the )
( user program at compile time )
: STORE            ( - ? | :=Ø ERROR; :=-1 }; :=1 O.K. )
              INSERT {
              BEGIN
                TRANSLATE  DUP
                IF    [ ' } CFA ] LITERAL =
                ENDIF
                GETOKEN
              UNTIL
              1          ;
( Store the semantics in the definition for execution at compile time )
: SEMANTICS
      SYMBOL  {
      IF      STORE  2 ?ERR
      ELSE    SYMBOL  {[
              IF      PERFORM 3 ?ERR
              ELSE    Ø
              ENDIF
      ENDIF   ;
```

```
( Parse a UNIT of a definition, including a cyclic structure )
: UNIT          NONTERMINAL -DUP
                IF -1 = IF HERE FORWARD ENDIF , 1
                ELSE STRING
                  IF TSTSTG 1
                  ELSE SYMBOL $
                    IF HERE SELF DROP BRANCHTRUE , INSERT SET 1
                    ELSE SYMBOL  (
                      IF [ HERE UNT' ! ] SELF 4 ?ERR DROP
                         SYMBOL ) 5 ?ERR
                         ELSE Ø
                      ENDIF
                    ENDIF
                  ENDIF
                ENDIF ;
( Parse a section of the alternative )
: SUBPHRASE     UNIT    IF      INSERT  ?CHKERR  1
                        ELSE    SEMANTICS
                        ENDIF   ;
( Parse an alternative )
: PHRASE        UNIT    IF      BRANCHFALSE    HERE Ø ,
                                BEGIN   SUBPHRASE
                                WHILE
                                REPEAT
                                HERE    SWAP    !
                                1
                        ELSE    SEMANTICS
                        ENDIF   ;
( Parse a definition )
: EXP   PHRASE  IF      SYMBOL  |
                        IF      BRANCHTRUE  HERE
                                Ø ,  SELF   6 ?ERR  DROP
                                HERE    SWAP    !
                        ENDIF
                        1
                ELSE    Ø
                ENDIF   ;
' EXP CFA UNT' @ ! ( Forward reference to EXP resolved )
( Enter a word into the vocabulary )
: MAKE          CURRENT @ CONTEXT !
                HERE    CONTEXT @ @ (FIND)
                IF      DROP NFA ID. 4 MESSAGE SPACE THEN
                HERE    DUP C@ WIDTH MIN 1+ ALLOT
                DUP 128 TOGGLE HERE  1 - 128 TOGGLE
                LATEST , CURRENT @ ! HERE 2+ ,
                DOCOL LATEST PFA CFA ! GETOKEN ;
( Complete the word definition )
: CLOSEDEF      [ ' ;S CFA ] LITERAL , ;
: INITIALIZE    1 -1 FTABLE              ( SET UP START ADDRESS )
( A )           SNUM SENTRY DUP +
( A S E*2 )     34 + * Ø
( A B Ø )       DO DUP I + Ø SWAP C! LOOP DROP ;
```

```
( Define a non-terminal )
: DEFINE          MAKE EXP CLOSEDEF DROP ;
( Define the main compiler )
: MAIN            MAKE
                  INSERT GETOKEN
                  INSERT MAKE
                  EXP
                  INSERT CLOSEDEF
                  CLOSEDEF
                  DROP ;
( Define a language )
: LANGUAGE        INITIALIZE      GETOKEN
                  BEGIN
                      SYMBOL  DEFINE
                      IF      DEFINE GETOKEN Ø
                      ELSE    SYMBOL  MAIN
                              IF      MAIN  1
                              ELSE    ." ILLEGAL KEYWORD" 1
                              ENDIF
                      ENDIF
                  UNTIL  RESOLVE ;
```