5-1-1995

The Design, construction, and implementation of an engineering software command processor and macro compiler

Jesse Coleman

THE DESIGN, CONSTRUCTION, AND IMPLEMENTATION OF AN ENGINEERING SOFTWARE COMMAND PROCESSOR AND MACRO COMPILER

by

Jesse J. Coleman

A Thesis Submitted
in
Partial Fulfillment
of the
Requirements for the

MASTER OF SCIENCE
in

Mechanical Engineering

Approved by:

Professor [Names Illegible]
Thesis Advisor

Professor_____

Professor_____

Professor_____
Department Head

DEPARTMENT OF MECHANICAL ENGINEERING
COLLEGE OF ENGINEERING
ROCHESTER INSTITUTE OF TECHNOLOGY

MAY 1995

Title of thesis THE DESIGN, CONSTRUCTION, AND IMPLEMENTATION OF AN ENGINEERING SOFTWARE COMMAND PROCESSOR AND MACRO COMPILER

I _____ hereby grant permission to the Wallace Memorial Library of RIT to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Date 5/10/95

ABSTRACT

This paper presents the design and construction of a software translator that serves as a foundation, or central engine, around which an entire engineering software system can be constructed. To provide the user with a powerful interface to drive an application, a high-level procedural language similar to FORTRAN or BASIC is integrated into the translator. An application shell was also written to provide the user with an interactive command line environment for using the translator. The translator, language, and application shell together mechanize a programming and command interpreter environment. Users can interactively enter commands from the keyboard or load and process prewritten macro files from disk. The language gives users the ability to create variables, arrays, and functions, process complex mathematical expressions, and develop sophisticated macro programs. The language is quite capable of solving very complex engineering problems. Several engineering examples are presented including a solution to a four-bar crank mechanism, adding a material library to an application, a command line integration solver, a Runge-Kutta routine for solving sets of differential equations, and a convolution integral routine. The translator is modular, easily extensible, written entirely in C/C++, and readily portable to different platforms. A set of diagnostic tools is integrated into the translator to aid the developer in future development work. Complete theory and design details for all phases of the translator and language are presented. Performance issues are studied including a comparison against C/C++ and MS-DOS Qbasic. Exploration in application integration for a simulation package similar to CSMP is investigated. A complete Language and Compiler Guide is supplied with the program.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

$\langle\ \rangle$	grammar nonterminal
ε	empty set; "null sequence" for a nonterminal
( )	grouping operator
[ ]	optional grammar segment, array operator
{ }	repeating grammar segment, set operator
|	"or" symbol; alternative choice
$\rightarrow$	grammar production symbol
θ, ϕ	angle (deg)
Q	heat transfer rate (Btu/hr)
T	temperature (°F), thrust force (lb_f), period (sec)
k	thermal conductivity (Btu/hr-ft-°F), spring constant (lb_f/ft)
A	surface area (ft^2)
Δx	distance (ft)
c_p	specific heat at constant pressure (Btu/lb_m-°F)
x, y	displacement (ft)
g	gravitational acceleration = 32.17 (ft/sec^2)
W	weight (lb_f)
K	aerodynamic drag coefficient (lb_f-sec^2/ft^2)
ζ	damping ratio
ω_0	natural frequency (rad/sec)
m	mass (slug)
c	damping coefficient (slug/sec)
h(t)	unit impulse response (ft/lb_f)
t	time (sec)
F, *f*	force (lb_f)

1. INTRODUCTION

Definition, Description, and Background

From practical experience and exposure to many different engineering software systems over the years, it's apparent that the most successful and versatile systems are the ones that incorporate a *command interpreter* and provide the user with some form of *programming environment*. A system of this type gives the end application user the most powerful means to interface with and drive an application. A typical command interpreter allows the user to interact with the application by entering commands at the keyboard or loading and processing prewritten macro files from disk. The inclusion of a high-level procedural language adds a programming environment that gives the user the ability to create variables, arrays, and functions, process complex mathematical expressions, and develop sophisticated macro programs. The ability to create and process macros is the key to a powerful and flexible system that allows a user to customize an application particular to their needs. Macros provide a powerful mechanism to support and drive an application:

- System configurations can be set or altered.
- Frequently used variables and functions can be loaded at startup and be immediately available to the user.
- Libraries of special functions can be created and loaded for use when needed. Libraries can transform a general purpose system into a specialized computing environment.
- Libraries of geometric entities or parts can be stored as macros and loaded when needed.
- Entire session files can be processed.
- Application processes can be automated.

To provide an application with a command interpreter and programming environment, a *translator* must be designed into the system. Technically, a translator is defined as a mechanism which takes as input a *source language* and produces as output either:

1) A generated *target language* for later execution on a host machine.
2) A generated target language which itself becomes the input source language for further translation.
3) *Interprets* during translation and *synthesizes actions* that are executed by calling appropriate host machine subroutines.

The process of translation involves *analysis* and *synthesis*. A translator must analyze a source language to determine it's content and structure, interpret the meaning or action intended by the language, and synthesize either a target language or directly execute host machine subroutines to perform the intended actions. A translator that generates a machine-oriented target language is called a *compiler*; a translator that interprets during translation and synthesizes actions is called an *interpreter*.

Translators allow the user to express algorithms and command statements in a high-level language independent of the host machine's target language. A FORTRAN compiler is good example of a translator used by engineers and scientists. The source language is tailored specifically for algorithmic, numerical computations and requires no knowledge of the host

machine language. The familiar DOS command language is another good example where a translator is used to implement a command interpreter. The source language consists of commands designed to interface with and drive the DOS operating system. System commands are entered at the keyboard, interpreted, and executed. The ability to write and process batch files adds a programming environment that gives the user the capability to customize and drive the DOS environment.

Translation mechanisms are present in one form or another in most applications. Even systems that do not have all the capabilities of the more advanced applications still implement some form of translation. The ability to simply key in a command requires rudimentary lexical, syntactic, and semantic analysis to be performed. Expression processing increases the complexity of translation and requires advanced parsing, type checking, implicit casting, and a full set of mathematical operators. The addition of user defined variables adds symbol table and management routines for memory allocation, insertion, searching, and deleting of variables. Language and translation theory even play important roles in areas one might not expect at first. Solid modelers that implement Constructive Solid Geometry (CSG) systems are based on the same language and parsing theory used to specify and translate grammars. Parse trees for a grammar parallel CSG trees for a solid. For instance, the primitives or tokens of a grammar (technically called terminals) might include FLOAT, LET, +, -; where CSG terminals would include CYLINDER, BLOCK, SPHERE, $\cap$ (intersection), $\cup$ (union), and – (difference). Grammar statements or sentences (called nonterminals) would include such items as *for_statement* and *if_statement*, while CSG nonterminals would represent intermediate parts that make up the final solid being constructed. Even the tools learned and used in translation development are indispensable in application development: linked lists, stacks, trees, etc. Linked lists are quite often used in applications to build dynamic data structures for objects such as geometric entities and to group parent/child relationships among objects. Quadtree and octree tree data structures are used by solid modelers to represent plane and solid geometric objects. FEA modelers imploy the same tree structures to implement mesh generation.

Whether directly or indirectly, translator theory and design is present in practically every engineering software application. Therefore, if one seriously endeavors to develop a professional engineering package; translator and language theory, design, and construction must be undertaken.

Developing a modern engineering software package is a major undertaking in many respects. To be competitive in today's market, an application must be designed to give the user as much flexibility and power as possible to interface with and drive an application. Graphical user interfaces manage forms, menus, and other widgets to provide a user-friendly environment that greatly increases ease of use and productivity. Interactive graphics allow the user to easily visualize, construct, and manipulate entities such as solid geometry, finite elements, boundary conditions, etc. However, as necessary and useful as these interfaces are, they do not afford the user the flexibility and power required to customize, extend, and drive the application. To accomplish this, an engineering software package must incorporate a translator to provide the application with a command interpreter and programming environment.

A superior system design includes a translator that provides:

1) An interactive command interpreter environment.

2) An application command set that gives the user access to all of the system routines and databases.

3) A high-level procedural language to support programming.

Many reasons lead to both the need for a translator and its construction first in the overall design of an application. From a user's point of view, a truly professional package should offer the following:

- Interactive command line input from the keyboard.
- The ability to process mathematical expressions.
- The ability to create variables and use them in expressions or as command arguments.
- The ability to write functions or macros to do calculations and drive application processes.
- A comprehensive application command set with access to system routines and databases.
- A high-level procedural language for programming.

From a developer's point of view, a translator offers the following advantages:

- No single system can be designed to meet all the requirements of every user. A programming and command interpreter environment gives the user as much flexibility as possible to customize and drive an application to meet their specific needs.
- A translator provides a solid foundation, or central engine, around which an entire application can be constructed.
- A system designed around a translator is easily extensible for future development.
- A programming and command interpreter environment is necessary to be competitive with the current state of the art in software packages.

Many successful systems that implement translators are in use today. The familiar PATRAN pre and post processor is a good example of a finite element system that includes a translator with a high-level procedural language and extensive application command set. The user can write functions or macros in PCL (Patran Command Language) and compile them into the system. These macros can then be used to construct geometry, generate meshes, model complex spatial and temporal fields, etc.

The CSMP (Continuous System Modeling Program) simulation program developed by IBM is a problem-oriented programming language that includes the FORTRAN language and a command set tailored towards solving ordinary differential equations and block diagram simulations. In actuality, CSMP is a translator that takes as input CSMP statements and produces as output a FORTRAN source language subroutine. This subroutine is then compiled with an integration routine using a standard FORTRAN compiler and executed.

The Hewlett-Packard ME10/30 Mechanical Engineering CAD Systems integrate a translator to provide a programming and command interpreter environment. This system incorporates a high-level procedural language and an extensive application command set that includes system commands to generate menus, control graphic viewports, interface to digital tablets, etc. As a matter of interest, it was this very system that was the main inspiration for this project.

Other examples include Unigraphics, AutoCAD, ACSL, ProENGINEER, NISA, Aries, etc.

Purpose

This paper presents the design and construction of a translator, language, and application shell that serve as a foundation and framework around which an entire engineering software system can be constructed. To provide the end application user with a powerful interface to drive an application, a high-level procedural language similar to FORTRAN or BASIC was developed and integrated into the translator. Since no application was actually constructed, an application shell was written to create an interactive command line environment for using the translator. The shell and translator comprise a single executable program: *cp.exe*. After system startup and initialization, a programming and command interpreter environment is created that lasts for the duration of the application session. Users can then interactively enter commands at the keyboard or load and process user-written macro files from disk.

Major features of the translator include:

- Creates a programming and command interpreter environment.
- Takes as input a fully structured source language very similar to BASIC or FORTRAN.
- Supports two modes of operation; an interpreter mode and a compile mode.
- Performs type checking and implicit casting.
- Extensive error checking is performed during all phases of translation including runtime support for floating point exceptions and array bounds checking.
- Reentrable; user-defined macros can call the translator to process commands or compile functions at runtime.
- Easily extensible for future development and application integration.
- Portable to other hardware platforms; the entire system was written in C/C++.
- A set of diagnostic tools is integrated into the translator to aid the developer.

The major features of the language include:

- Free format of source language text.
- Symbolic constants.
- Variables and multi-dimensional arrays.
- User-defined types or records.
- User-defined functions that take parameters and return values.
- Program flow control constructs; e.g., IF, SWITCH, FOR, REPEAT, WHILE, etc.
- Local, global, and external objects.
- Mathematical functions; e.g., SIN(x), ABS(x), COSH(x), LN(x), etc.
- Complex arithmetic expressions; e.g., x = EXP(-zeta*wn*(t-s))*SIN(SQRT(1.0-zeta^2.0)*wn*(t-s)).
- Screen and disk I/O functions; e.g., OPEN, CLOSE, PRINT, PRINT #, INPUT, INPUT #, etc.

The project resulted in the following major items:

- Complete theory and design details for all phases of the translator and language.
- *CP.EXE* - An executable translator program that creates a programming and command interpreter environment. The program includes an integrated application shell to provide the user with an interactive command line environment.
- A complete Language and Compiler Guide.
- A set of diagnostic tools to aid the developer.
- Translator C++ source code modules and several example macro files.
- Example code solving engineering problems; Four-Bar Crank Mechanism, Material Library, Integration Solver, Runge-Kutta DEQ Solver, and Convolution Integral.
- Performance evaluation including a comparison against C/C++ and MS-DOS QBasic.
- Exploration in application integration for a simulation package similar to CSMP.

2. TRANSLATOR THEORY

2.1 Overview

A *translator* is a mechanism which takes as input a *source language* and produces as output either:

1) A generated *target language* for later execution on a host machine.

2) A generated target language which itself becomes the input source language for further translation.

3) *Interprets* during translation and *synthesizes actions* that are executed by calling appropriate host machine subroutines.

The process of translation involves *analysis* and *synthesis*. A translator must analyze a source language to determine it's content and structure, interpret the meaning or action intended by the language, and synthesize either a target language or directly execute host machine subroutines to perform the intended actions. A translator that generates a machine-oriented target language is called a *compiler*; a translator that interprets during translation and synthesizes actions is called an *interpreter*.

Conceptually, translation may be grouped into *phases* as shown below:

- Lexical Analysis — The process of constructing valid language tokens from the input source language text stream.
- Syntax Analysis — The process of recognizing whether a given token sequence is a valid language statement, and if so, what its structure is.
- Semantic Analysis — The process of performing the actions or meaning intended by the language.
- Intermediate Code Generation — The production of an intermediate form of the source language that is easily optimized and translatable into a target language or executable by an interpreter.
- Code Generation — Generation of target language.
  or
  Interpretation — Execution of appropriate host machine subroutines to perform the intended actions of the source language.

Additional phases of translation include:

- Static Checking — The many types of checks that are performed during translation such as type checking, flow control checking, uniqueness checking, etc.
- Code Optimization — The process of optimizing the intermediate code to produce faster running target code.
- Symbol Table Management — Management of the data structures used by the translator to store and retrieve records of objects and their associated attributes.
- Error Management — Error tracking, reporting, and recovery management during all phases of the translation process.

Conceptually, phases are distinct processes performed during translation. Depending on the type of translator design, phases are often combined together and some even omitted such as code optimization. Translator designs are commonly structured into modules; each module's job to process a phase of group of phases.

The phases of translation can also be grouped into *front* and *back ends*. The front end typically includes all phases of translation necessary to process the source language into an intermediate language form. The back end includes all phases necessary to process the intermediate language into executable code or synthesize actions by interpretation. A typical compiler or interpreter layout grouped into front and back ends is shown in Figure 2.1.

Quite a bit of theory has been developed around the phases of translation and include studies into many areas such as formal grammars, finite automata, parsing techniques, error recovery and correction, data-flow analysis, etc. This paper presents in the next sections an introduction to some of the basic theory relevant to this particular translator design. The reader is referred to any of the translator books given in the Bibliography section of this paper for further background on translator theory and methods.

The following major topics will be covered:

- Context-Free Grammars
- Lexical Analysis
- Syntax Analysis using Recursive Descent with Predictive Parsing
- Syntax-Directed Translation with Embedded Semantic Processing
- Threaded Code Interpreters

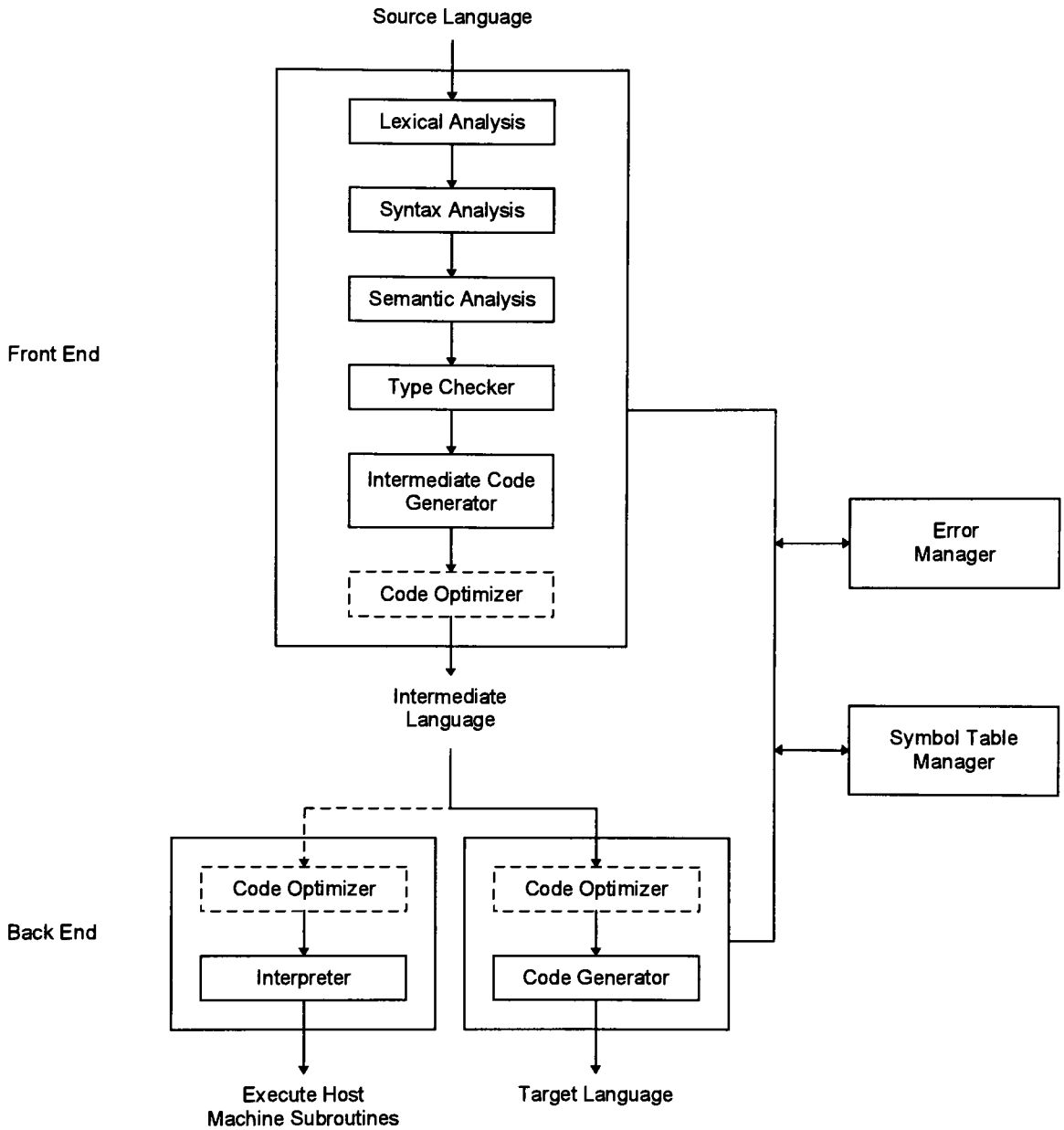


Figure 2.1 Typical compiler/interpreter translation phases.

2.2 Grammars

A *grammar* is a formal specification of the structure, or *syntax*, of a language. A grammar defines what symbols are used to write a language and what sequences of symbols are permitted.

2.2.1 Context-Free Grammars

A *context-free grammar* is a type of grammar often used to describe the syntactic structure of programming languages.[1]

A context-free grammar consists of four parts:

1. A finite, nonempty set of symbols called *terminals*. Terminals are the words and symbols that make up the language; e.g., **FOR**, **+**, *****, **{**, **LET**, **WHILE**, etc.

2. A finite, nonempty set of *nonterminals*. Nonterminals represent sequences of terminals and/or nonterminals used in a language; e.g., *for_statement*, *assignment_statement*, *if_statement*, *expression*, etc.

   For example, a WHILE block is represented by the nonterminal *while_statement* which represents the sequence **WHILE(***expr***)** *statement_list* **ENDWHILE**.

   WHILE, **(**, **)**, and **ENDWHILE** are terminals, *expr* and *statement_list* are nonterminals.

3. A finite, nonempty set of rules called *productions*. A production associates the replacement of a single nonterminal with a sequence of terminals and/or nonterminals or the null sequence, ε.

   For example, the production for a *while_statement* would be given by:

   while_statement → **WHILE(***expr***)** *statement_list* **ENDWHILE**

   If alternative productions for a nonterminal exist, each is grouped with the nonterminal and preceded by the production symbol "→". Alternatively, the "|" symbol may be used to indicate an alternative production.

   For example, the following sets of productions for *expr* are equivalent:

   expr → *term* + *term*
   → *term* - *term*

   expr → *term* + | - *term*

4. A designation of one of the nonterminals in part 2 as a *starting nonterminal* from which all others can be generated by systematic application of the rules in part 3.

To illustrate the concept of a context-free grammar, a simple example for evaluating expressions is presented in Example 2.1.

Example 2.1. A context-free grammar specification for evaluating expressions is given below:

1. terminal set: { **number**, **(**, **)**, **+**, **-**, **/**, ***** }
2. nonterminal set: {*expr*,*term*,*factor*}
3. productions:

   a) *expr* → *term* + *term*
   → *term* - *term*
   → *term*

   b) *term* → *factor* * *factor*
   → *factor* / *factor*
   → *factor*

   c) *factor* → **(** *expr* **)**
   → **number**

4. Starting nonterminal: *expr*

Now consider the expression 3/5+(8-2)*4. To show this is a valid expression in the language, a sequence of productions applied to *expr* using the rules in part 3 must exist. The application of rules to derive the final form consisting of all terminals is called a *derivation*. Application of rules on *expr* to derive 3/5+(8-2)*4 is shown below in Table 2.1.

RULE	PRODUCTION
3.a	*expr* → *term* + *term*
3.b	*expr* → *factor* / *factor* + *term*
3.c	*expr* → 3 / *factor* + *term*
3.c	*expr* → 3 / 5 + *term*
3.b	*expr* → 3 / 5 + *factor* * *factor*
3.c	*expr* → 3 / 5 + (*expr*) * *factor*
3.a	*expr* → 3 / 5 + (*term* - *term*) * *factor*
3.b	*expr* → 3 / 5 + (*factor* - *term*) * *factor*
3.c	*expr* → 3 / 5 + (8 - *term*) * *factor*
3.b	*expr* → 3 / 5 + (8 - *factor*) * *factor*
3.c	*expr* → 3 / 5 + (8 - 2) * *factor*
3.c	*expr* → 3 / 5 + (8 - 2) * 4

Table 2.1 Grammar productions to derive an expression.

The derivation used in Example 2.1 to generate the expression is not unique. However, for an unambiguous grammar, there exists a unique left-most and right-most derivation. The derivation shown was a *left-most* derivation. A left-most derivation systematically applies rules to derive nonterminals from left to right. A *right-most* derivation does the same but from right to left.

A grammar that produces more than one left-most or right-most derivation is termed ambiguous. Ambiguous grammars have more than one left or right-most derivation and special handling is required by the parser to resolve the ambiguity.

The parser implemented in the translator implements left-most derivation for all cases except exponentiation in expressions. Exponentiation has right to left associativity and a right-most derivation is used in that case.

High-level language constructs are just as easily represented by a context-free grammar. Shown below are some typical productions for a few of the common flow control constructs.

A typical WHILE block:

while_statement → **WHILE(** *expr* **)** *statement_list* **ENDWHILE**

A simple IF block:

if_statement → **IF(** *expr* **)** *statement_list* **ENDIF**

A powerful form of the FOR block:

for_statement → **FOR(** *assignment_list* **;** *expr* **;** *assignment_list* **)** *statement_list* **NEXT**

An entire program can be the starting symbol for a grammar with a production as shown below:

program → **PROGRAM** *program_name statement_list* **END_PROGRAM**

2.2.2 Parse Trees

A parse tree is a graphical representation of the productions applied to a starting nonterminal to reach the final terminal symbols of the grammar. A parse tree has the following characteristics:

1. A single starting node, or root, which is the starting nonterminal.
2. Leaves that are terminal symbols.
3. Interior nodes that are nonterminals.

A parse tree for the expression given in Example 2.1 is shown in Figure 2.2.

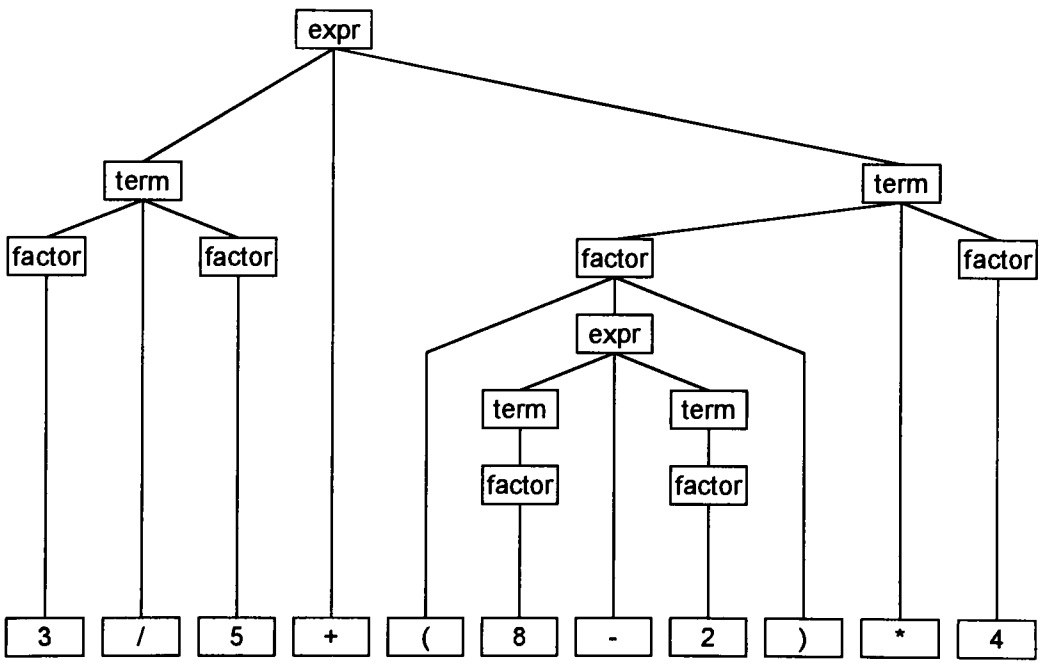


Figure 2.2 Parse tree for a simple expression.

The derivation given in Example 2.1 also illustrates a *depth-first* traversal of the parse tree. A depth-first traversal descends down the parse tree to reach the leaves as quickly as possible.

2.3 Syntax Analysis

Syntax analysis, or *parsing*, is the process of recognizing whether a given string is in a language, and if so, what its structure is. Parsing reads input terminals from left to right, and using the grammar of the language, attempts to construct the parse tree for the derivation. Two major classes of parsing techniques exist; top-down and bottom-up.

Top-down parsing starts at the root, or starting nonterminal, and builds downward to the leaves. Parsing systematically applies productions to single nonterminals to yield sequences of terminals and/or nonterminals until the terminals, or leaves of the tree, are reached.

Bottom-up parsing starts at the leaves of the tree and builds upward to the root. Parsing systematically substitutes sequences of terminals and/or nonterminals for single nonterminals until the starting nonterminal, or root of the tree, is reached

Much work has gone into the theory of parsing and as a result, many techniques exist for implementing parser schemes. The reader is referred to any of the compiler books given in the Bibliography section of this paper for background on parsing methods.

2.3.1 Recursive Descent with Predictive Parsing

The parsing method chosen for use in this translator is a popular top-down method called *recursive descent* with *predictive parsing*. This technique is easily extendible, uses the activation records of function calls to implicitly build a parse tree, exploits function recursion, requires no backtracking (the next token read always determines which routine to call), and directly parallels the format of the grammar. Semantic actions and type checking are easily integrated into the parsing routine. Additionally, for someone without a background in compiler theory, it's the easiest technique to understand and learn.

Major elements of a Recursive Descent Predictive Parser

- A procedure or routine exists for each nonterminal in the grammar.
- The next token to be read, called the *lookahead symbol*, unambiguously determines which procedure or routine to call.
- The construction of a parse tree is implicit in the function call activation records.
- The parser exploits function recursion for nesting of statements and expressions.

To illustrate predictive parsing using the recursive descent technique, see Example 2.2.

Example 2.2 Consider the production for the while statement:

while_statement → **WHILE(** *expr* **)** *statement_list* **ENDWHILE**

A typical 'C' routine to process this production would be as follows:

```
void while_statement()
{
 match(WHILE); match('('); expr(); match(')'); statementlist(); match(ENDWHILE);
}
```

match() is a short routine that calls the lexical analyzer to get the next token. It takes a parameter that is compared to the current token; if they match, the next token is extracted, if they don't match, an error is generated. The parameter passed to match() is actually predicting what the current token must be according to the grammar specification.

expr() is the main routine to evaluate expressions for the nonterminal *expr*.

statementlist() is the main routine to evaluate statements for the nonterminal *statement_list*.

To simulate the parsing process, assume some routine has already extracted the next token and found it to be **WHILE**. This triggers a call to the function while_statement().

Step 1. Current token is **WHILE**. Call match(WHILE) to extract the next token.

Step 2. Current token is either **(** or something else. Call match('('); if the current token is **(**, the next token is extracted, if not, an error is generated.

Step 3. Call routine expr(). This routine will process the expression. If successful, no errors will be generated and program control returns.

Step 4. Current token is either **)** or something else. Call match(')'); if the current token is **)**, the next token is extracted, if not, an error is generated.

Step 5. Call routine statementlist(). This routine will process the statements in the body of the while block. Since while blocks can be nested, this routine may actually call while_statement() several times, which in turn calls statementlist(). This is where the recursive nature of function calls is used to implement nesting. If successful, no errors will be generated and program control returns.

Step 6. Current token is either **ENDWHILE** or something else. Call match(ENDWHILE); if the current token is **ENDWHILE**, the next token is extracted, if not, an error is generated.

Step 7. The while statement has been successfully parsed, return to calling routine.

2.3.2 Note on Left Recursion

Left recursion occurs when the left-most nonterminal on the right side of a production is the same nonterminal as on the left side of the production. For example,

expr → *expr* + *term*

This form of production will cause the recursive descent parser to go into an infinite loop making repeated calls to expr(). Left recursion can be eliminated by introducing a new nonterminal and rewriting the grammar to be right recursive in form.

expr → *term more_terms*

more_terms → + *term*

→ ε

For a general discussion on left recursion elimination, see Aho, Sethi, and Ullman.[2]

2.4 Semantic Analysis

The process of parsing discussed in the previous section carries out recognition and syntactic analysis of the source language. *Semantic analysis* performs the actions or meaning intended by the language. *Semantic rules* define the actions required to carry out the activities necessary for translation; e.g., insertion of objects into a symbol table, intermediate code generation, error handling, etc.

Syntax-directed definitions and *translation schemes* relate semantic rules with productions. Both are a generalization of a context free grammar where each grammar symbol has an associated set of attributes; *synthesized* and *inherited*. Inherited attributes are computed from the parent and siblings of a node in the parse tree. Synthesized attributes are computed from the attributes of the children of a node in the parse tree.

Listed below are some of the major elements of each:

Syntax-Directed Definitions

- High-level specifications for translations.
- Hide implementation details and order of evaluation.
- Dependency graphs and topological sorting to determine evaluation order of semantic rules.
- Syntax trees and dags to decouple translation from parsing.

Translation Schemes

- Indicate order in which semantic rules are to be evaluated.
- Reveal implementation details.
- Semantic actions are coupled with parsing.

The reader is referred to Aho, Sethi, and Ullman for a detailed discussion on syntax-directed definitions and translation schemes.[3]

2.5 Syntax-Directed Translation

A *syntax-directed translation scheme* was implemented in the translator to generate postfix expressions for the entire language. A syntax-directed translation scheme associates *attributes* with each element of the grammar and embeds semantic processing with parsing. Attributes can be anything; a value, a memory address, an object type, etc. The productions of the grammar are modified to include semantic actions where necessary to implement the semantic rules.

For example, let's look again at Example 2.1 and this time include semantic analysis to generate an intermediate language string in postfix notation. Table 2.2 shows the productions and syntax-directed definitions for the language.

PRODUCTIONS			SEMANTIC RULES
expr	→	*term + term*	$expr.val = term_1.val \mid\mid term_2.val \mid\mid$ '+'
	→	*term - term*	$expr.val = term_1.val \mid\mid term_2.val \mid\mid$ '-'
	→	*term*	*expr.val = term.val*
term	→	*factor * factor*	$term.val = factor_1.val \mid\mid factor_2.val \mid\mid$ '*'
	→	*factor / factor*	$term.val = factor_1.val \mid\mid factor_2.val \mid\mid$ '/'
	→	*factor*	*term.val = factor.val*
factor	→	(*expr*)	*factor.val = expr.val*
	→	**number**	*factor.val* = **number**.*val*

Table 2.2 Grammar productions and syntax-directed definitions.

The semantic rules in Table 2.2 associate synthesized string attributes for each production. The || symbol represents string concatenation. The derivation of an expression will result in a final string representation of the output in *expr.val*. To implement a translation scheme, we first define an action symbol { }. An action symbol is used to represent a semantic action in a production. For example, *expr* → term + term { emit('+') }. Here a routine called emit is called to output the '+' symbol.

Embedding semantic actions in the grammar yields the following:

expr → *term* + *term* { emit('+') }
→ *term* - *term* { emit('-') }
→ *term*

term → *factor* * *factor* { emit('*') }
→ *factor* / *factor* { emit('/') }
→ *factor*

factor → (*expr*)
→ **number** { emit(**number**) }

Note that the translation scheme presented above emits strings as the productions are parsed. This differs from the syntax-directed definition given in Table 2.2 that synthesizes strings through concatenation until the final string is stored in *expr.val*.

Now, again consider the expression 3/5+(8-2)*4. Processing of the expression will result in the following intermediate string representation: 3 5 / 8 2 - 4 * +

Application of rules on *expr* to derive 3/5+(8-2)*4 with embedded semantic actions is shown below in Table 2.3.

RULE	PRODUCTION	SEMANTIC ACTION		
3.a	*expr* → *term* + *term*			
3.b	*expr* → *factor* / *factor* + *term*			
3.c	*expr* → 3 / *factor* + *term*	3		
3.c	*expr* → 3 / 5 + *term*	5	/	
3.b	*expr* → 3 / 5 + *factor* * *factor*			
3.c	*expr* → 3 / 5 + (*expr*) * *factor*			
3.a	*expr* → 3 / 5 + (*term* - *term*) * *factor*			
3.b	*expr* → 3 / 5 + (*factor* - *term*) * *factor*			
3.c	*expr* → 3 / 5 + (8 - *term*) * *factor*	8		
3.b	*expr* → 3 / 5 + (8 - *factor*) * *factor*			
3.c	*expr* → 3 / 5 + (8 - 2) * *factor*	2	-	
3.c	*expr* → 3 / 5 + (8 - 2) * 4	4	*	+

Table 2.3 Grammar productions with embedded semantic actions.

Let's again revisit Example 2.2 but include semantic processing.

Example 2.3 Consider the production of the while statement with embedded semantic actions.

while_statement → {inc blocklevel}
{emit("while")}
WHILE (*expr*) {emit("if")}*statement_list* **ENDWHILE** {emit("wend")}
{emit{breakcount), emit("setbrk")}
{dec blocklevel}

Semantic actions:

blocklevel	Integer representing the level or nested depth of the current while statement.
while	Keyword for back end compiler indicating start of while block.
if	Keyword for back end compiler to insert code to evaluate *expr* and jump to appropriate location.
wend	Keyword for back end compiler indicating end of while block.
breakcount	Integer representing number of BREAK statements in current while block.
setbrk	Keyword for back end compiler to insert jump code for break statements.

The actual source code to process a while block is shown below. LP is a global pointer to a linked list of level records. Levels correspond to the depth of nested flow control statements. Any flow control statement creates a new level record on entry and inserts it in the list. LP always points to the most recent record which represents the current level. The BREAK statement uses LP to store a count of the number of breaks that occur in the current level. On exit, the current level record is deleted and LP is set to point to the next higher up level.

```
static void while_statement()
{
 if(error.status()) return;

 LevelRecord *Level = new LevelRecord;
 Level->Next = LP;
 LP = Level;

 match(WHILE); emit("while"); match('('); boolexpr(); match(')'); emit("if");
 statementlist(); match(ENDWHILE); emit("wend");
 emit("ii"); emit(itoa(LP->breakcnt,AddressBuffer,10)); emit("setbrk");

 LP = Level->Next; delete Level;
}
```

Processing a single while block with no nesting and two break statements would result in the following intermediate code generation:

while ...*expression code*... if ...*statement code*... wend ii 2 setbrk

2.6 Lexical Analysis

Lexical analysis is the process of constructing valid language tokens from the input source text stream. Lexical analysis can be included in the grammar specification for parsing or separated into a separate phase.

For example, a portion of a context-free grammar to construct the nonterminals ⟨*identifier*⟩ and ⟨*number*⟩ is presented below.

⟨*identifier*⟩	→	⟨*letter*⟩ {⟨*letter*⟩ | ⟨*digit*⟩}
⟨*number*⟩	→	⟨*integer*⟩ ["**.**"] [⟨*exponent*⟩]
	→	[⟨*integer*⟩] "**.**" ⟨*integer*⟩ [⟨*exponent*⟩]
⟨*integer*⟩	→	⟨*digit*⟩ {⟨*digit*⟩}
⟨*exponent*⟩	→	"**E**" | "**e**" [⟨*sign*⟩] ⟨*integer*⟩
⟨*sign*⟩	→	"**+**" | "**-**"
⟨*digit*⟩	→	"**0**" | "**1**" | "**2**" | ... | "**9**"
⟨*letter*⟩	→	"**A**" | "**B**" | ... | "**Z**" | | "**_**" | | "**a**" | "**b**" | ... | "**z**"

Of course, some routine is needed to extract characters from the source text. Also, note that tokens are not in the terminal set of the grammar in this implementation, but appear as nonterminals.

More commonly, lexical analysis is handled by a separate phase of the translator. The lexical analyzer, or "*scanner*", is called by the parser to extract the next token from the input source text stream. This implementation allows classification of tokens and inclusion of tokens into the grammar's terminal set. White space, comment removal, and error handling are also much easier to implement. A *state table* mechanizes the construction of tokens and is discussed in detail in the design section of this paper.

2.7 Static Checking

Static checking includes many types of checks that must be done by the compiler during translation. These checks include:

Type Checking:

Assuring that operands are compatible with each other, operations are defined for a particular operand type, function parameters are of the appropriate type, assignments are compatible, etc.

Flow Control Checking:

Checking that flow control statements such as BREAK and RETURN appear in legal context in a source code program. For example, RETURN cannot be used outside of a function definition and must be used if a function is to return a value.

Uniqueness Checking:

Checking that external objects have been declared only once, objects declared in the body of a function are unique, function parameters have distinct names, etc. For example, suppose an external variable is declared as FLOAT xcoor. A check must exist to search the symbol table to be sure that xcoor has not been previously declared as something else.

Many other checks may have to be done depending on the complexity, structure, and content of the language and design of the translator. Most checks are easily embedded into the translation scheme.

2.8 Intermediate Code Generation

Although a translator can be designed to directly generate target code from a source language, intermediate code generation is commonly employed in translator designs. Intermediate code generation has the following advantages:

- Different back end translators can be designed to produce target code for various host machines and still use the same front end translator.
- A single code optimizer phase can be designed to optimize the intermediate form independent of the host machine.
- A more easily optimized and translatable form of language can be generated for target code production or execution by an interpreter.

Three common forms of intermediate representation are syntax trees, postfix notation, and three-address code. Postfix notation was chosen for this design to allow the inclusion of a back end threaded interpreter/compiler which is itself a stack-based, postfix notation translator.

Postfix notation requires the operands of an expression precede the operator. For example, the infix expression 4+5/3 would have the postfix form: 4 5 3 / +. Extending this notation to include function calls, a call such as hyp(x,y,z) would have the postfix form: x y z hyp.

Postfix notation is easily implemented using a stack based machine which will be discussed in detail in the threaded code interpreter section of this paper. Intermediate language generation is easily implemented in a syntax-directed translation scheme.

2.9 Code Optimization

Code optimization is an attempt to perform transformations on code to ultimately produce performance improvements. Optimization can be applied at several stages:

- Source Code — User applies code improvements to source code by improving algorithms, etc.
- Intermediate Code — Optimizing phase of compiler performs local and global optimizing techniques such as subexpression elimination, copy propagation, dead-code elimination, loop optimization, etc.
- Target Code — Optimizing phase of compiler performs optimization techniques to take advantage of particular host machine's hardware such as register usage, instruction set, etc.

No optimization phase is currently implemented in the translator. For a detailed discussion on code optimization, see Aho, Sethi, and Ullman.[4]

2.10 Code Generation

The generation of target code from intermediate language code is the final phase of the compiling translator. Typical target code includes absolute or relocatable machine language or assembly language. For a detailed discussion on code generation, see Aho, Sethi, and Ullman.[5]

The translator designed for this project does not actually generate machine-dependent code, but instead, generates a *threaded code* (list of addresses that point to host machine subroutines.) This threaded code is processed by a software interpreter which will be discussed in detail in the sections that follow.

2.11 Interpretation

Interpretation synthesizes actions during translation that are executed by calling appropriate host machine subroutines. Interpreters can be software or hardware in design. A typical computer executing a program is actually running a hardware interpreter built into the CPU. The main steps involved are:

1. Fetch the next instruction pointed to by the instruction pointer IP
2. Increment the instruction pointer IP
3. Decode the instruction
4. Execute the instruction
5. Repeat steps 1-4

Of course, the CPU interpreter is processing a list of binary machine instructions specific to the host machine. A software interpreter, on the other hand, usually processes a list of abstract machine instructions which have a corresponding host machine language subroutine associated with them. The steps involved in a software interpreter are very similar:

1. Fetch the next abstract machine instruction pointed to by the instruction pointer
2. Increment the abstract machine instruction pointer
3. Determine which host machine subroutine is to be called
4. Execute the host machine subroutine
5. Repeat steps 1-4

A threaded code interpreter is a form of software interpreter which was chosen for this translator design and is discussed in detail in the next section.

2.12 Threaded Code Interpreter/Compiler

A *threaded code interpreter* is itself a translator that incorporates a threaded code generator and software interpreter. Threaded code is a fully analyzed internal form of instructions comprised of addresses that point to either:

Primitives which are executable host machine subroutines or

Secondaries that are lists of threaded code instructions pointing to primitives and/or secondaries.

The job of the interpreter is to step thru a list of threaded code instructions and execute the code associated with each instruction. If the instruction points to a primitive, a call to the host machine subroutine is made and executed. If the instruction points to a secondary, which itself is a list of threaded code instructions, the interpreter's instruction counter is set to point at the new list, and the interpreter processes the secondary's list of instructions. After the secondary has been processed, control returns back to the next instruction following the call to process the secondary.

Primitives point to the actual code that's executed by the host machine. Secondaries group pointers to primitives and other secondaries and form a list of threaded code instructions akin to a program, function, or subroutine. The ability of the threaded code interpreter to create secondaries is very similar to the action of a compiler generating a machine code target language. Because of this, the threaded code interpreter can be considered to have two modes of operation; a *compile mode* where secondaries are created and an *execution mode* where threaded code is interpreted and executed.

The term *Threaded Interpretive Language*, or *TIL* for short, refers to the internal form of code generated by the translator. However, it's commonly used in reference to all elements of the interpreter, and will be likewise adopted here for convenience.

The main elements of a TIL are:

- Primitives and Secondaries
- Dictionary
- Stacks and Registers
- Outer Interpreter
- Inner Interpreter

The following sections describe the main elements of the TIL. The interpreter sections will pull all the elements together and should give a clear picture of the mechanization and operation of the threaded interpreter.

2.12.1 Primitives and Secondaries

Primitives and secondaries are called *keywords*, much the same as reserved words in other languages. Each have distinct record structures as shown in Figure 2.3.

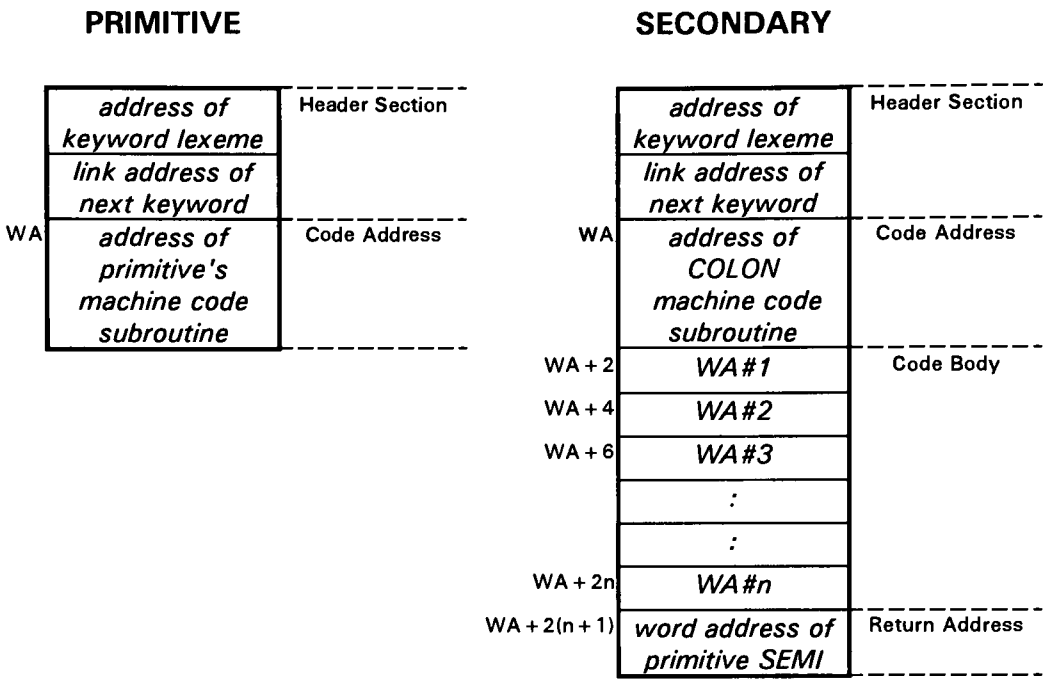


Figure 2.3 Primitive and secondary code structures.

Each keyword is comprised of sections as follows:

- *Header* Address of the keyword's name or lexeme and the link address of the next keyword in the dictionary. Note that some system keywords exist that have no headers. These are not available to the user but are used by the translator to perform special operations. Since the translator is aware of their existence and location, no header is required. For example, SEMI is a headerless primitive inserted at the end of each secondary by the compiler.

- *Code Address* Address of an executable machine code subroutine. For a primitive, the machine code subroutine is the actual machine code to execute whatever function the primitive was designed to do. For a secondary, the machine code routine is always COLON, a special subroutine designed into the interpreter to handle secondaries.

- *Code Body* For a primitive, the actual machine code routine located somewhere in memory (not shown in Figure 2.3.) For a secondary, the code body section is a list of threaded code word addresses; WA#1, WA#2, WA#3, ... Each WA# corresponds to the starting word address, WA, of a primitive or secondary.
- *Return Address* Word address of the primitive SEMI which is used to terminate the execution of a secondary.

Assuming a 2-byte word address scheme, all primitives with headers have a fixed record size of 6 bytes. Secondaries have variable record lengths depending on the number of threaded code instructions that are compiled into the code body of the secondary.

In addition to primitives and secondaries, threaded code lists may also include addresses that are not instructions, but instead, are pointers to literals such as numbers, character strings, and the like. *Literal handlers* are primitives designed to allow for the inclusion of such literals in threaded code lists. For each literal, the word address of the appropriate literal handler must immediately precede the address of the literal in the threaded code list. During execution, the literal handler will extract the literal at the address currently pointed to by the instruction pointer, push it to the data stack, and then increment the instruction pointer past the literal address to the next valid instruction in the threaded code list.

2.12.2 Dictionary

The TIL maintains a data structure called a *dictionary*. The dictionary is a linked list of TIL keyword records. At startup, the dictionary is loaded with all the system primitives. As translation proceeds, the dictionary dynamically grows with the insertion of new keywords as secondaries are created in compile mode. See Figure 2.4.

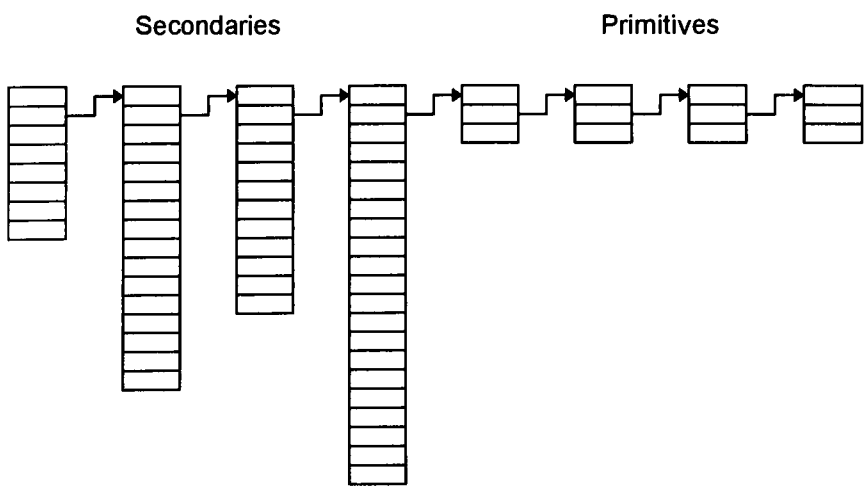


Figure 2.4 TIL dictionary.

2.12.3 Stacks and Registers

The TIL interpreter uses stacks and several registers as described below:

DS	Data Stack; LIFO stack used to store numbers and addresses
RS	Return Stack; LIFO stack used to store return addresses when a secondary calls another secondary or a primitive.
I_reg	Instruction Register; used to store address of next threaded code instruction in current secondary being processed.
WA_reg	Word Address Register; used to store the word address of the current keyword or the address of the code body section of the current keyword.
CA_reg	Code Address Register; used to store address of next executable machine code body.

2.12.4 Outer Interpreter

The outer interpreter serves as a controlling executive for the interpreter. In many TIL's, the outer controlling executive is itself a secondary written in the language of the TIL.

The outer interpreter, just as other translators, performs rudimentary lexical, syntax, and semantic analysis, as well as calling the inner interpreter to generate threaded code or execute instructions.

The basic operation of the outer interpreter is as follows:

Step 1. Scan the input text stream and extract the next token.

Step 2. If the token is an address or number,
- a) If execute mode, push it to the data stack, repeat Step 1.
- b) If compile mode, insert it in the threaded code list of the current secondary under construction, repeat Step 1.

Step 3. Search the dictionary to see if the token is a keyword. If found,
- a) If execute mode, push the word address WA of the keyword into the word address register and execute the inner interpreter, repeat Step 1.
- b) If compile mode, insert the keyword word address WA into the threaded code list of the current secondary under construction, repeat Step 1.

Step 4. Token is not a number, address, or keyword; generate an error.

2.12.5 Inner Interpreter

The inner interpreter mechanizes the language by executing a small loop as illustrated in the pseudo code shown below:

```
REPEAT
  MOV *WA_reg, CA_reg
  CALL CA_reg
  MOV *I_reg, WA_reg
  I_reg++
UNTIL(EXITFLAG==TRUE)
```

To start the process prior to entering the loop, the word address, WA, of the current keyword to be executed is copied into the WA_reg. EXITFLAG is set FALSE, and the WA of a special headerless entry to set EXITFLAG to TRUE is copied into I_reg. This preparation sets up the current keyword to be executed and also adds an instruction to call an exit routine after the keyword has been executed.

Upon entering the loop, the followings steps are performed:

1) Copy the address of the machine code routine pointed to by WA_reg into the code address register, CA_reg.

   MOV *WA_reg, CA_reg

2) Execute the machine code routine at the address in CA_reg.

   CALL CA_reg

3) Copy the word address pointed to by the next threaded code instruction in I_reg to the word address register WA_reg.

   MOV *I_reg, WA_reg

4) Increment the address in the instruction register I_reg. This is the address of the next threaded code instruction.

   I_reg++

5) Check EXITFLAG. If TRUE, exit the inner interpreter loop. If FALSE, repeat steps 1) thru 4).

Two supporting machine code routines exists to mechanize the processing of secondaries; COLON and SEMI. These routines perform the following actions:

COLON Push the contents of the instruction register I_reg to the return stack RS. Increment the WA_reg, then copy the contents of word address register WA_reg to the instruction register I_reg.

```
PUSH I_reg, RS
WA_reg++
MOV WA_reg, I_reg
```

SEMI Pop the top address off the return stack RS, copy it into the instruction register I_reg.

```
POP   RS, I_reg
```

To best illustrate the operation of the interpreter, a step by step analysis of the processing of an assignment statement is given in the following example.

Example 2.4 Processing an assignment statement in execute mode.

Assume the front end translator has processed the following source language assignment statement:

x = 3.0 + pi2

and emitted the intermediate language stream:

296068 3.0 pi2 f+ !f

where 296068 is the address of the variable x, pi2 is a secondary that returns 2*pi, f+ is a primitive to add two floating point numbers, and !f is a primitive to store a floating point number at a specified address.

Two secondaries have been previously created, pi and **pi2**. The keyword **pi** is a simple routine that pushes the value 3.14159 to the data stack. The keyword **pi2** is another simple routine that calls **pi** twice, then adds the two top stack values together to leave 6.28319 on the data stack. Simulated dictionary entries for the two secondaries, along with all the supporting primitives and machine code routines, is shown in Figure 2.5.

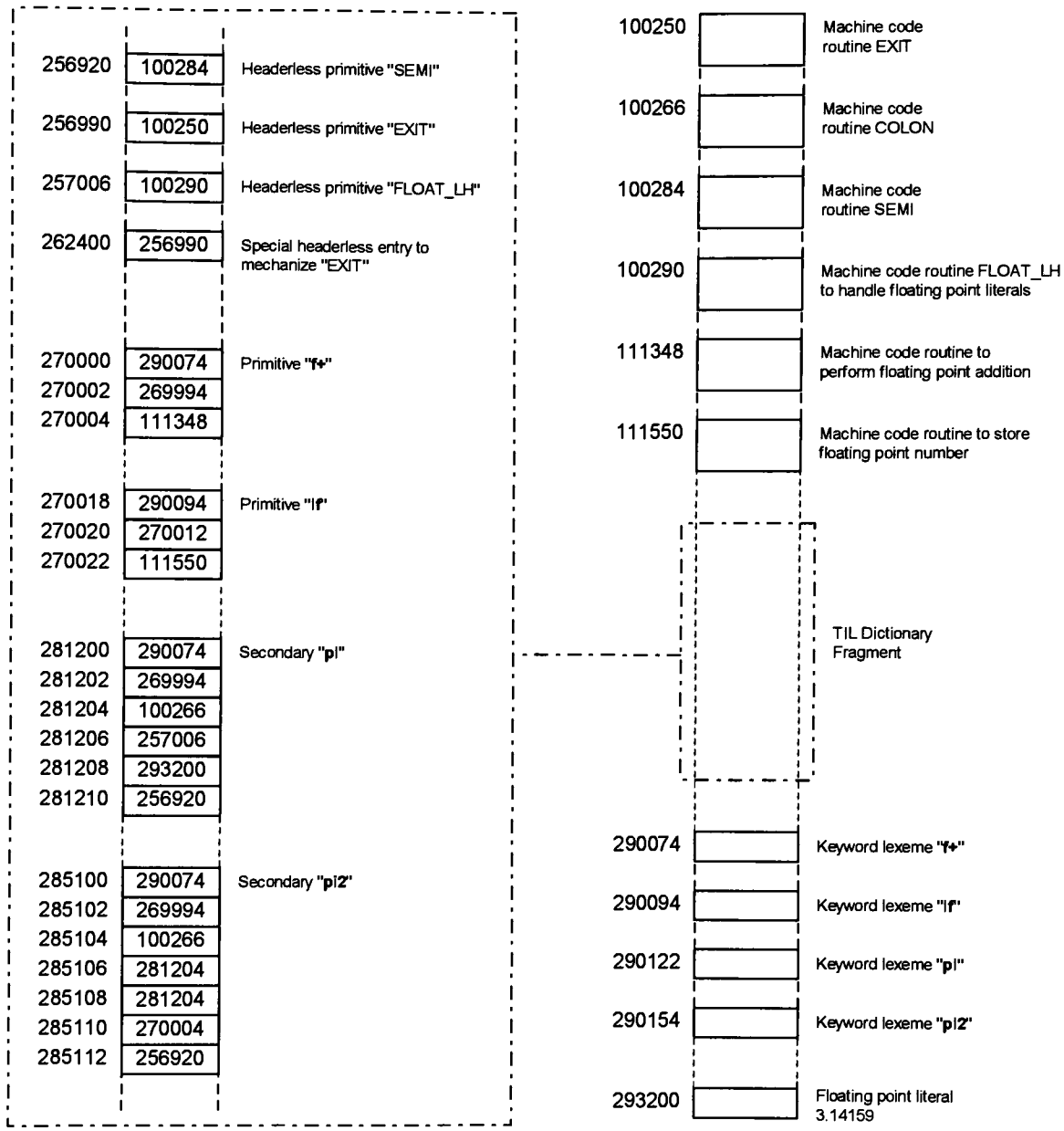
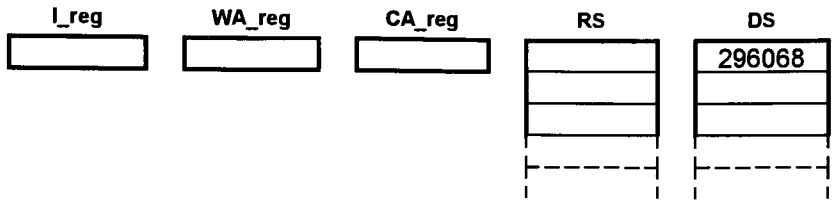


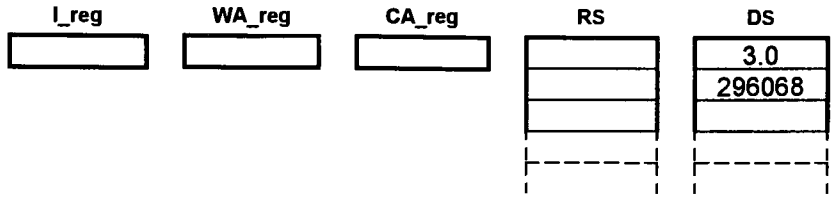
Figure 2.5 Memory layout of TIL dictionary and machine code routines.

Assuming the TIL is in execute mode, the outer interpreter will perform the following actions:

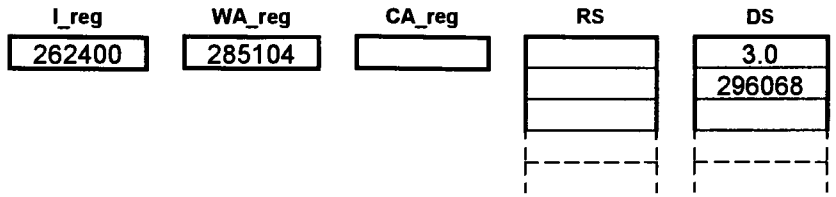
1) Extract the address 296068 of variable x and push it to the data stack



2) Extract the number 3.0 and push it to the data stack



3) Lookup the keyword "**pi2**" in the TIL dictionary, copy the word address WA of the keyword into the word address register WA_reg, copy the word address of the headerless entry to implement EXIT into the instruction register I_reg, and execute the inner interpreter.



The entire step by step action of the interpreter to process the assignment statement is shown in Table 2.4.

Routine	Instruction	I_reg	WA_reg	CA_reg	RS	DS
OUTER	PUSH 296068, DS					296068
	PUSH 3.0, DS					3.0, 296068
	PUSH 285104, WA_reg		285104			3.0, 296068
	PUSH 262400, I_reg	262400	285104			3.0, 296068
INNER	MOV *WA_reg, CA_reg	262400	285104	100266		3.0, 296068
	CALL CA_reg	262400	285104	100266		3.0, 296068
COLON	PUSH I_reg, RS	262400	285104	100266	262400	3.0, 296068
	WA_reg++	262400	285106	100266	262400	3.0, 296068
	MOV WA_reg, I_reg	285106	285106	100266	262400	3.0, 296068
INNER	MOV *I_reg, WA_reg	285106	281204	100266	262400	3.0, 296068
	I_reg++	285108	281204	100266	262400	3.0, 296068
	MOV *WA_reg, CA_reg	285108	281204	100266	262400	3.0, 296068
	CALL CA_reg	285108	281204	100266	262400	3.0, 296068
COLON	PUSH I_reg, RS	285108	281204	100266	285108, 262400	3.0, 296068
	WA_reg++	285108	281206	100266	285108, 262400	3.0, 296068
	MOV WA_reg, I_reg	281206	281206	100266	285108, 262400	3.0, 296068
INNER	MOV *I_reg, WA_reg	281206	257006	100266	285108, 262400	3.0, 296068
	I_reg++	281208	257006	100266	285108, 262400	3.0, 296068
	MOV *WA_reg, CA_reg	281208	257006	100290	285108, 262400	3.0, 296068
	CALL CA_reg	281208	257006	100290	285108, 262400	3.0, 296068
FLOAT_LH	PUSH **I_reg, DS	281208	257006	100290	285108, 262400	3.14159, 3.0, 296068
	I_reg++	281210	257006	100290	285108, 262400	3.14159, 3.0, 296068
INNER	MOV *I_reg, WA_reg	281210	256920	100290	285108, 262400	3.14159, 3.0, 296068
	I_reg++	281212	256920	100290	285108, 262400	3.14159, 3.0, 296068
	MOV *WA_reg, CA_reg	281212	256920	100284	285108, 262400	3.14159, 3.0, 296068
	CALL CA_reg	281212	256920	100284	285108, 262400	3.14159, 3.0, 296068
SEMI	POP RS, I_reg	285108	256920	100284	262400	3.14159, 3.0, 296068
INNER	MOV *I_reg, WA_reg	285108	281204	100284	262400	3.14159, 3.0, 296068
	I_reg++	285110	281204	100284	262400	3.14159, 3.0, 296068
	MOV *WA_reg, CA_reg	285110	281204	100266	262400	3.14159, 3.0, 296068
	CALL CA_reg	285110	281204	100266	262400	3.14159, 3.0, 296068
COLON	PUSH I_reg, RS	285110	281204	100266	265110, 262400	3.14159, 3.0, 296068
	WA_reg++	285110	281206	100266	285110, 262400	3.14159, 3.0, 296068
	MOV WA_reg, I_reg	281206	281206	100266	285110, 262400	3.14159, 3.0, 296068
INNER	MOV *I_reg, WA_reg	281206	257006	100266	285110, 262400	3.14159, 3.0, 296068
	I_reg++	281208	257006	100266	285110, 262400	3.14159, 3.0, 296068
	MOV *WA_reg, CA_reg	281206	257006	100290	285110, 262400	3.14159, 3.0, 296068
	CALL CA_reg	281208	257006	100290	285110, 262400	3.14159, 3.0, 296068
FLOAT_LH	PUSH **I_reg, DS	281208	257006	100290	285110, 262400	3.14159, 3.14159, 3.0, 296068
	I_reg++	281210	257006	100290	285110, 262400	3.14159, 3.14159, 3.0, 296068
INNER	MOV *I_reg, WA_reg	281210	256920	100290	285110, 262400	3.14159, 3.14159, 3.0, 296068
	I_reg++	281212	256920	100290	285110, 262400	3.14159, 3.14159, 3.0, 296068
	MOV *WA_reg, CA_reg	281212	256920	100284	285110, 262400	3.14159, 3.14159, 3.0, 296068
	CALL CA_reg	281212	256920	100284	285110, 262400	3.14159, 3.14159, 3.0, 296068

Table 2.4 Step by step action of TIL interpreter.

Routine	Instruction	I_reg	WA_reg	CA_reg	RS	DS
SEMI	POP RS, I_reg	285110	256920	100284	262400	3.14159, 3.14159, 3.0, 296068
INNER	MOV *I_reg, WA_reg	285110	270004	100284	262400	3.14159, 3.14159, 3.0, 296068
	I_reg++	285112	270004	100284	262400	3.14159, 3.14159, 3.0, 296068
	MOV *WA_reg, CA_reg	285112	270004	111348	262400	3.14159, 3.14159, 3.0, 296068
	CALL CA_reg	285112	270004	111348	262400	3.14159, 3.14159, 3.0, 296068
FADD	PUSH ADD(POP DS,POP DS),DS	285112	270004	111348	262400	6.28319, 3.0, 296068
INNER	MOV *I_reg, WA_reg	285112	256920	111348	262400	6.28319, 3.0, 296068
	I_reg++	285114	256920	111348	262400	6.28319, 3.0, 296068
	MOV *WA_reg, CA_reg	285114	256920	100284	262400	6.28319, 3.0, 296068
	CALL CA_reg	285114	256920	100284	262400	6.28319, 3.0, 296068
SEMI	POP RS, I_reg	262400	256920	100284		6.28319, 3.0, 296068
INNER	MOV *I_reg, WA_reg	262400	256990	100284		6.28319, 3.0, 296068
	I_reg++	262402	256990	100284		6.28319, 3.0, 296068
	MOV *WA_reg, CA_reg	262402	256990	100250		6.28319, 3.0, 296068
	CALL CA_reg	262402	256990	100250		6.28319, 3.0, 296068
EXIT	MOV TRUE, EXITFLAG	262402	256990	100250		6.28319, 3.0, 296068
INNER	MOV *I_reg, WA_reg	262402	######	100250		6.28319, 3.0, 296068
	I_reg++	262404	######	100250		6.28319, 3.0, 296068
OUTER	PUSH 270004, WA_reg	262404	270004	100250		6.28319, 3.0, 296068
	PUSH 262400, I_reg	262400	270004	100250		6.28319, 3.0, 296068
INNER	MOV *WA_reg, CA_reg	262400	270004	111348		6.28319, 3.0, 296068
	CALL CA_reg	262400	270004	111348		6.28319, 3.0, 296068
FADD	PUSH ADD(POP DS,POP DS),DS	262400	270004	111348		9.28319, 296068
INNER	MOV *I_reg, WA_reg	262400	256990	111348		9.28319, 296068
	I_reg++	262402	256990	111348		9.28319, 296068
	MOV *WA_reg, CA_reg	262402	256990	100250		9.28319, 296068
	CALL CA_reg	262402	256990	100250		9.28319, 296068
EXIT	MOV TRUE, EXITFLAG	262402	256990	100250		9.28319, 296068
INNER	MOV *I_reg, WA_reg	262402	######	100250		9.28319, 296068
	I_reg++	262404	######	100250		9.28319, 296068
OUTER	PUSH 270022, WA_reg	262404	270022	100250		9.28319, 296068
	PUSH 262400, I_reg	262400	270022	100250		9.28319, 296068
INNER	MOV *WA_reg, CA_reg	262400	270022	111550		9.28319, 296068
	CALL CA_reg	262400	270022	111550		9.28319, 296068
STORE	MOV(POP DS, POP DS)	262400	270022	111550		
INNER	MOV *I_reg, WA_reg	262400	256990	111550		
	I_reg++	262402	256990	111550		
	MOV *WA_reg, CA_reg	262402	256990	100250		
	CALL CA_reg	262402	256990	100250		
EXIT	MOV TRUE, EXITFLAG	262402	256990	100250		
INNER	MOV *I_reg, WA_reg	262402	######	100250		
	I_reg++	262404	######	100250		
OUTER	Exit outer interpreter					

Table 2.4 (cont.) Step by step action of TIL interpreter.

3. TRANSLATOR DESIGN

3.1 Overview

The main goal of this project was to design a translator with an easily extensible command language for future development and include a core set of high-level language constructs typical of modern procedural languages. Two separate audiences were considered in the design of the language and translator; the *application developer* and the *end application user*.

The application developer will need to interface an application command set into the language at a future point in time. To provide ease of extensibility for the developer, the following language, translator, and development schemes were implemented:

- Context-free grammar
- Syntax directed translation scheme
- Recursive descent parser
- Modular design of translator
- C/C++ development language; platform independent design

The end application user will require all the power of a high-level language to interface with and drive an application. To achieve this, the following core language constructs were designed into the system:

- Variables and multidimensional arrays.
- User defined types or records.
- Functions that take parameters and return values.
- Flow control constructs; e.g., IF, SWITCH, FOR, REPEAT, WHILE, etc.
- Local, global, and external objects.
- Mathematical functions; e.g., SIN(x), ABS(x), COSH(x), LN(x), etc.
- Complex arithmetic expressions
- Free format
- Symbolic constants

The translator design for this project consists of a front end translator incorporating a recursive descent parser and a back end threaded interpreter/compiler. See Figure 3.1.

The *front end* translator:

- Takes as input a fully structured source language very similar to BASIC or FORTRAN.
- A top down recursive descent parsing technique is used to implement a syntax-directed translation scheme for infix to postfix notation with embedded semantic actions.
- Type checking and implicit casting are integrated into the parser.
- A separate lexical analyzer handles the task of token construction, classification, and attribute binding.
- All storage is static; symbol table management and memory allocation tasks are handled by the front end.
- Extensive error checking is performed during all phases of translation.
- The intermediate language generated is in postfix notation, type checked, and syntactically correct in form.

The *back end* threaded interpreter/compiler:

- A threaded code interpreter very similar to FORTH.
- Takes as input a postfix language consisting of numbers, addresses, and keywords.
- Two modes of operation; an interpreter mode and a compile mode.
- In compile mode, keyword definitions are compiled into threaded code to become new keywords in the language.
- In interpreter mode, keywords are executed immediately.
- A software interpreter very similar to the actual hardware interpreter in a computer is mechanized to execute the threaded code.
- Floating point exceptions and runtime array bounds checking are supported.

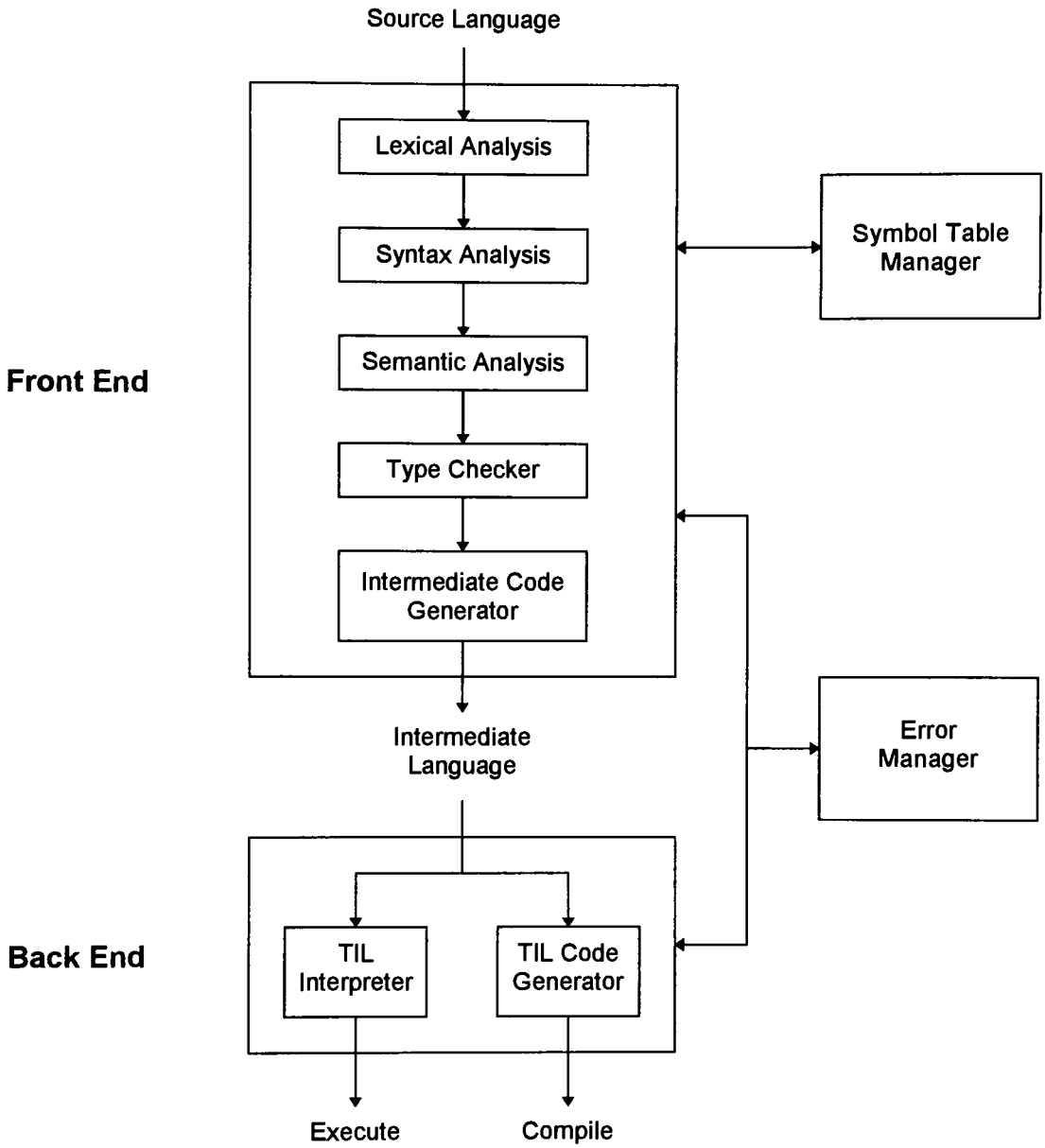
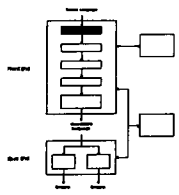


Figure 3.1 Actual translator design for this project.

3.2 Lexical Analysis



The lexical analysis phase of the compiler has the task of constructing valid language tokens from the input source text stream. The lexical analyzer, or "scanner", is called by the parser to extract the next token from the input source text stream.

The lexical analyzer performs the following tasks:

- Character recognition and classification.
- Token construction and classification.
- Removal of comments and white space from the source code.
- Error checking.

Character recognition and token construction are separated into two distinct processes as shown in Figure 3.2. The *scanner* handles the job of token construction; the *transliterator* extracts characters from the input stream buffer.

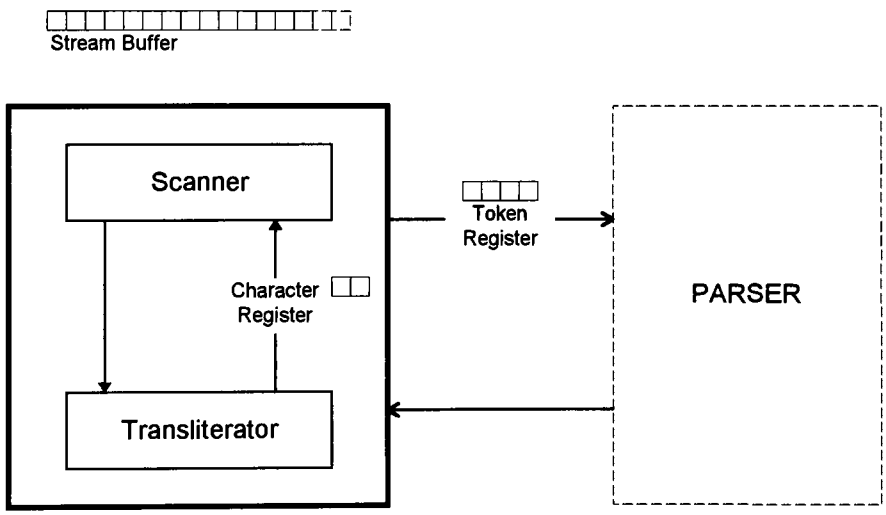


Figure 3.2 Lexical analyzer.

3.2.1 Transliterator

The function of the transliterator is to extract the next character from the input stream buffer, classify it, and return a class and attribute value for the character. Classification simplifies the task of the scanner during token construction. For example, to construct an integer, the scanner only needs to know the character is a digit between 0 and 9, but not the exact digit. Attributes encode additional information about an object; in this case, the ascii code of the character. A simple table lookup mechanism is implemented to handle character recognition and classification.

3.2.1.1 Character Classification

Character classifications and attributes are shown in Table 3.1.

CLASS	*Description*	*Characters*	VALUE (ASCII code)
DIGIT	numbers	0 thru 9	48-57
LETTER	letters and underscore	A thru Z, _, a thru z	65-90, 95, 97-122
ARITHMETIC	arithmetic operators	*, +, -, /, ^	42,43,45,47,94
RELATIONAL	relational operators	<, >, =	60-62
NEWLINE	line feed	(LF)	10
WS	white space	(HT, VT, FF, CR, space)	9,11-13,32
CONTROL	control characters		0-31,127 excluding 'WS'
ILLEGAL	upper ascii characters		128-255
NONE	all others in range ASCII 32-126	#, %, &, [, ...	35, 37, 38, 91, ...

Table 3.1 Character classification table.

3.2.1.2 The Character Register

A *character register* is used to store the current character's class and attribute value. Classes are enumerated constants, each having a unique integer value. Attribute values are simply the corresponding ascii code for the character. For example, if LETTER=2 and the current character is the letter 'F', then the character register contents returned to the scanner would be:

	CLASS	VALUE
CHARACTER REGISTER	2	70

3.2.1.3 The Stream Buffer

The source text stream is actually stored in a buffer as shown in Figure 3.3. Two pointers are used to navigate the buffer; BP0 and BP. BPO is always set to the start of the current character on entrance to the lexical analyzer. This marks the start of the current token being constructed. BP is used to mark the next character to be read. On entrance to the lexical analyzer, BP=BP0.

In operation, the transliterator extracts the character pointed to by BP, then increments BP to point at the next character in the buffer. As characters are extracted, BP is incremented. When the last character read causes the recognition of a token, BP0 and BP mark the token in the buffer and are used to extract the token lexeme. On exit, BP0 is set to BP to mark the start of the next token.

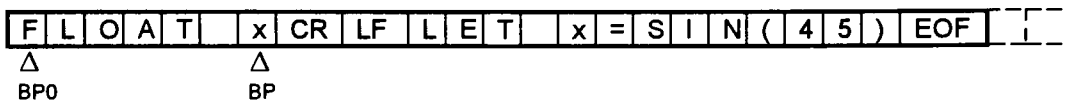


Figure 3.3 Stream buffer and pointers.

The scanner module frequently deincrements BP when a token is recognized. This effectively pushes back the last character read that caused the recognition of a token, but was not a part of the token itself. This is necessary so that when the parser calls for the next token, the character will not be skipped. For example, in Figure 3.3, the space character caused the recognition of the token FLOAT. When the transliterator extracted the space, BP was incremented to point at the next character, x. The scanner must deincrement BP so that when the lexical analyzer is called again, space will be the current character in the buffer.

3.2.2 Scanner

The function of the scanner is to construct a valid token from the characters it receives from the transliterator, classify it, resolve token attributes, and return it to the parser. Comments and white space are also stripped from the source text during this phase. A state table is implemented to mechanize token recognition and classification.

3.2.2.1 Token Classification

Token classifications and attributes are shown in Table 3.2.

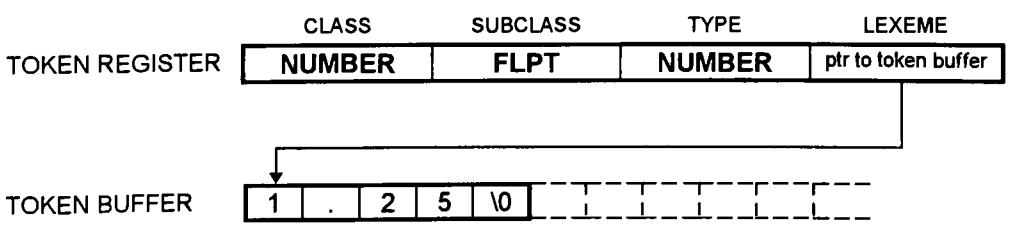
CLASS	*Description*	SUBCLASS	TYPE	LEXEME
IDENTIFIER	identifier	IDENTIFIER, VARIABLE, ARRAY, OR FUNCTION	IDENTIFIER, VARIABLE, ARRAY, OR FUNCTION	pointer to lexeme
RESERVED	reserved word	RESERVED	FLOAT, IF, REPEAT, etc...	pointer to lexeme
TYPE	type specifier	type #	TYPE	pointer to lexeme
NUMBER	real or integer number	FLPT, INT	NUMBER	pointer to lexeme
SLITERAL	string literal	SLITERAL	SLITERAL	pointer to lexeme
SPECIAL	operator	SPECIAL	ascii # of single character or LT, LTE, GT, GTE, etc...	
DONE	end of stream	DONE	DONE	

Table 3.2 Token classification table.

3.2.2.2 The Token Register

Once the scanner recognizes a token, a copy of the token is extracted from the stream buffer and placed in a token buffer. A *token register* is used to store the current token's class and attribute values. Classes are enumerated constants, each having a unique integer value. Attributes depend on the class of token, but can include Subclass, Type, and Lexeme.

For example, suppose the current token is the floating point number **1.25**. The contents of the token register and buffer would be:



Some additional examples are presented in Table 3.3 to clarify the contents of the token register for different classes of tokens. Bold uppercase names in the token register represent enumerated constants having unique integer values. Actual numbers are the unique integer codes for the current token, if applicable.

CURRENT TOKEN			TOKEN REGISTER			
DESCRIPTION	TOKEN	CODE	CLASS	SUBCLASS	TYPE	LEXEME
New identifier	ycoor		**IDENTIFIER**	**IDENTIFIER**	**IDENTIFIER**	ptr to token buffer
Existing variable identifier	**xcoor**		**IDENTIFIER**	**VARIABLE**	**VARIABLE**	ptr to token buffer
Existing array identifier	**A2**		**IDENTIFIER**	**ARRAY**	**ARRAY**	ptr to token buffer
Reserved word	**WHILE**	301	**RESERVED**	**RESERVED**	**301**	ptr to token buffer
User defined type	**POINT**	328	**TYPE**	**328**	**TYPE**	ptr to token buffer
Integer number	**37**		**NUMBER**	**INT**	**NUMBER**	ptr to token buffer
String literal	**"Material"**		**SLITERAL**	**SLITERAL**	**SLITERAL**	ptr to token buffer
Single character operator	**+**	43	**SPECIAL**	**SPECIAL**	**43**	
Multi-character operator	**<=**	128	**SPECIAL**	**SPECIAL**	**128**	
End of stream	**EOF**	-1	**DONE**	**DONE**	**DONE**	

Table 3.3 Examples of token classifications.

When the scanner recognizes a token as being an identifier, it must search the symbol table to determine if the identifier already exists as a variable, array, or function, or whether the token is a reserved word or type. This task could be handled by the parser, but was included in the lexical analyzer to resolve all token attributes in one phase.

3.2.2.3 State Diagrams

To construct a state table for a given language, *state diagrams* are used. A state diagram pictorially represents the states and transition paths a machine would step through to construct a token. A state diagram has a starting state, transition paths, intermediate states, and a recognition state. For example, to construct an identifier given a starting state S with the next character being a letter, see Figure 3.4.

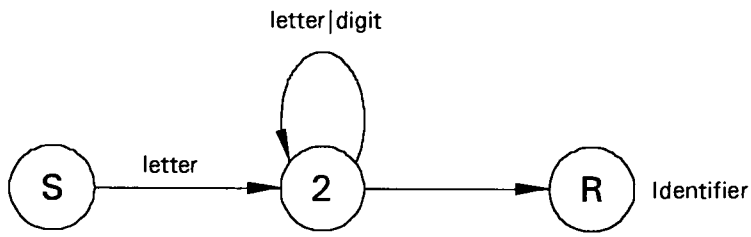


Figure 3.4 State diagram to construct an identifier.

Starting at state S, a letter is received. The system moves to state 2 along the transition path. Once at state 2, the system now has two transition paths it can follow. Receiving a letter or digit, the system moves again to state 2. The process continues until neither a letter or digit is received at which time the system moves to the recognition state R; an identifier.

A more complex state diagram to construct an unsigned number is shown in Figure 3.5.

3.2.2.4 State Table

Once the state diagrams have been constructed, a state table can be created as shown in Table 3.4. Each row represents a state and each column represents a transition path corresponding to the character class or value returned from the transliterator. Numbers in the table correspond to intermediate states. Recognition, starting, and error states are subscripted in the table to refer to the listing and descriptions given below the table.

For example, given a starting state S and transition path LETTER, we see from the table that the next state would be state 2. Once at state 2, there are 3 possible transition paths; DIGIT, LETTER, or other. DIGIT or LETTER again return to state 2 to continue the process of building an identifier. Any other transition path leads to the recognition state R3; an identifier.

A state table can be mechanized quite easily using a switch block in 'C'.

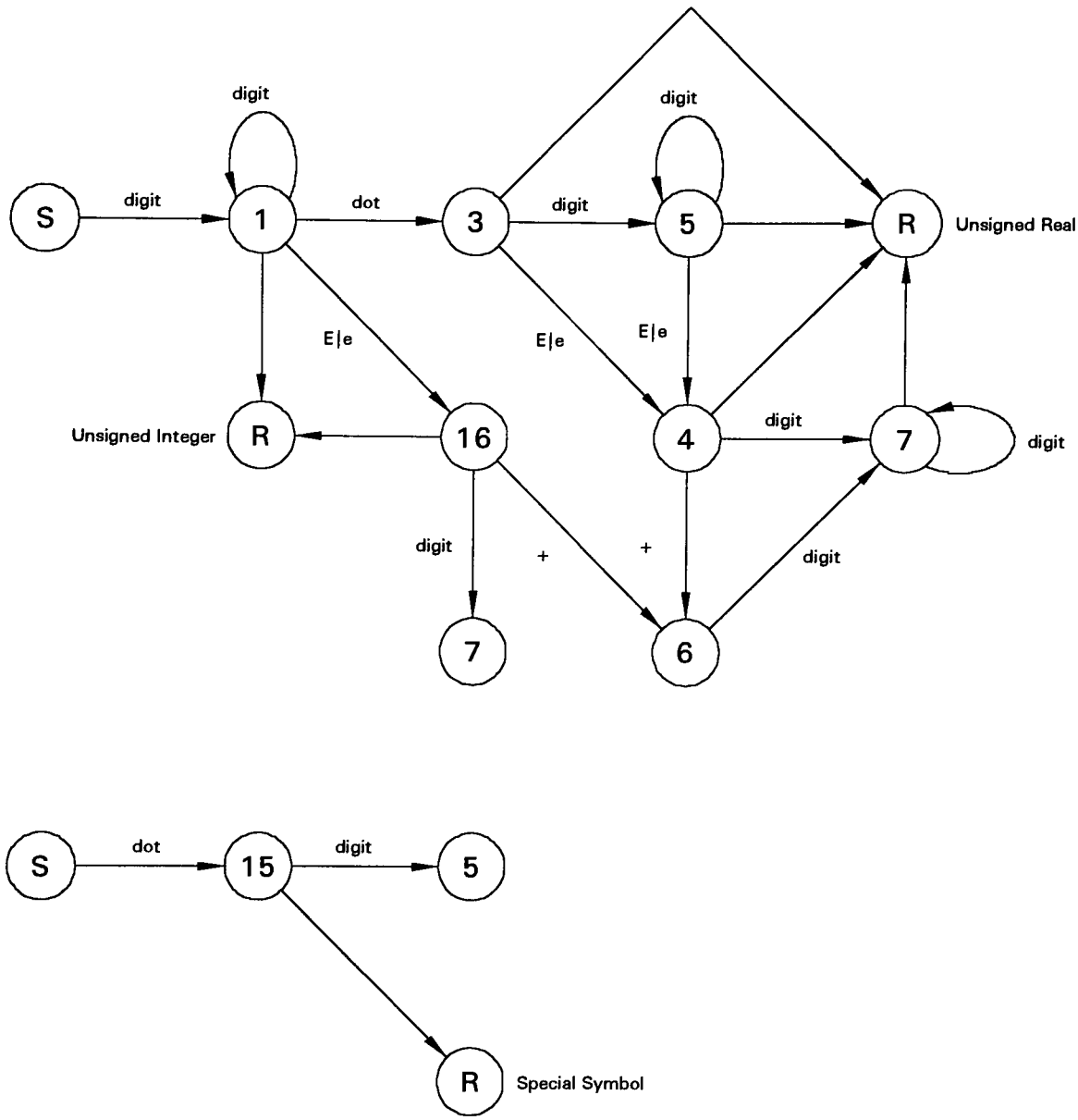


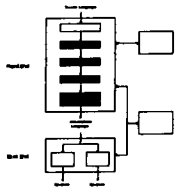
Figure 3.5 State diagram to construct an unsigned number.

	DIGIT	LETTER		/	SPACE	"	E	e	+		*	NEWLINE	EOF	=	<	>	ILLEGAL	other
S	1	2	15	8	13	14						S1	R1		17	18	ER1	12
1	1		3				18	18										R2
2	2	2																R3
3	5						4	4										R4
4	7								8	8								R5
5	5						4	4										R8
8	7																	ER2
7	7																	R7
8				11							9							12
9											10	9	ER3					9
10				S2							10	9	ER4					9
11												S3	R8					11
12																		R9
13					13													S4
14						R10							ER5					14
15	5																	12
18	7								8	8								R11
17														R12		R13		12
18														R14				12
ER																		R15

S1	newline character
S2	end of multiline comment
S3	end of single line comment
S4	stripping out white space complete
R1	end of file reached
R2 R11	integer number
R3	identifier
R4 R5 R8 R7	floating point number
R8	end of file and single line comment
R9	special character
ER1	illegal character
ER2	missing exponent
ER3 ER4	end of file before end of comment
ER5	end of file before end of string literal
R10	end of string literal
R12	special multicharacter <=
R13	special multicharacter <>
R14	special multicharacter >=
R15	an error has occurred

Table 3.4 State table.

3.3 The Parser



The syntax analysis phase of the compiler has the task of parsing an input token string to determine whether the string is grammatically correct, and if so, what it's structure is. Embedded semantic actions carry out the activities necessary for translation. Together, syntax and semantic analysis constitute a syntax-directed translation scheme. Type checking and intermediate code generation are also embedded in the translation scheme. These modules collectively are referred to as the *parser*. See Figure 3.6.

The parser performs the following tasks:

- Grammar recognition and structure.
- Semantic actions; e.g., symbol table management, nested block level tracking, implicit casting, etc.
- Type checking.
- Error management.
- Intermediate code generation.

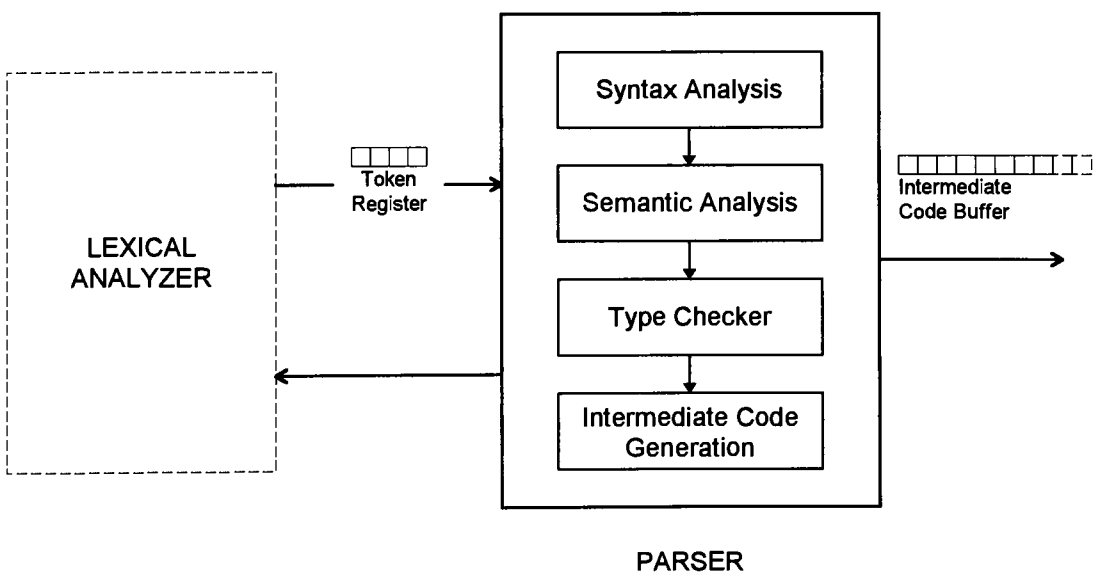


Figure 3.6 Parser.

3.3.1 Syntax Analysis With Embedded Semantic Actions

Syntax analysis is accomplished using a recursive descent technique with predictive parsing. Recall from the theory section of this paper, that for each nonterminal in the grammar, a corresponding function is written. These functions closely parallel the grammar specification of the language. For example, the highest level of abstraction in the grammar is at the root of the parse tree, the *command stream*. The command stream is a collection of objects; command statements and function definitions. Shown below is the grammar specification for a command stream.

⟨***command_stream***⟩	→	⟨*object*⟩ ⟨*more_objects*⟩
⟨*more_objects*⟩	→	⟨*object*⟩ ⟨*more_objects*⟩
	→	ε
⟨*object*⟩	→	⟨*function_definition*⟩
	→	⟨*command_statement*⟩
	→	ε

A typical function to implement this portion of the grammar is shown below.

```
:
:
match(Token.Type);
rdparser();
:
:
void rdparser()
{
  while(Token.Type!=DONE)
  {
    switch(Token.Type)
    {
      case(DEFINE):   function_definition();    break;
      default:        command_statement(); break;
    }
  }
}
```

Prior to executing this function, a call is made to the lexical analyzer to get the first token. The token's class and attributes are stored in the global token register, **Token**. The while loop mechanizes the collection of objects until the stream buffer is completely consumed and the lexical analyzer returns a token attribute of Token.Type=**DONE**. In the body of the while loop, a switch block is implemented to determine which type of object is currently being processed. If the token's attribute Token.Type=**DEFINE**, the object is the start of a function definition and a call is made to a routine to process function definitions. Otherwise, its assumed the object is a command statement and a call is made to a routine to process command statements.

The code example given above did not have any embedded semantic actions or error management. A more realistic translation routine that embeds error handling is illustrated below:

```
void rdparser()
{
  while(!error.status() && Token.Type!=DONE)
  {
    switch(Token.Type)
    {
      case(DEFINE):   function_definition();   break;
      default:        command_statement(); break;
    }
    if(error.status()) return;
  }
}
```

A collection of routines exist to handle errors in the translator. When an error occurs, a call is made to register the error using error.set(). To see if an error has been registered, a call to error.status() is made. Error checking is embedded in the above code at each iteration of the while loop and after each call to process a function definition or command statement. If an error has occurred, translation terminates and program control returns to the function that called rdparser(). At that point, appropriate code would be executed to display the error and reset the translator.

The code presented above will successfully parse a command stream to determine if it is syntactically correct. However, semantic actions must be included to effect the translation as shown below.

```
void rdparser()
{
  while(!error.status() && Token.Type!=DONE)
  {
    switch(Token.Type)
    {
      case(DEFINE):   function_definition();   break;
      default:        command_statement(); break;
    }
    if(error.status()) return;
    emit(NULL);
    til(BR->bufout);
    BR->bpout = BR->bufout;
  }
}
```

The intermediate code generated by the parser is written to a buffer by calling the routine emit(). The processing of a command statement or function definition will usually result in several calls to emit() to generate the intermediate language stream. Upon returning to rdparser(), the buffer is null terminated with a call to **emit(NULL)**. Next, a call is made to the back end translator to process the intermediate language stream; **til(BR->bufout)**, where BR->bufout is a pointer to the start of the intermediate code buffer. Finally, the pointer **BR->bpout**, which is used to navigate the output buffer, is reset to point to the start of the output buffer.

Of course, not shown above are the thousands of lines of source code and hundreds of functions which actually comprise all the nonterminals in the language that make up command statements and function definitions.

Along with the many functions that define the grammar for the language, the parser consists of the following main supporting routines, registers, and buffers:

- match() A short routine that calls the lexical analyzer to get the next token. It takes a parameter that is compared to the current token; if they match, the next token is extracted, if they don't match, an error is generated. Quite often, the parameter passed to match() is simply the current token. This forces a match and gets the next token. However, at times, the parameter passed to match() is actually predicting what the current token must be according to the grammar specification. If they don't match, an error is generated.

- emit() Routine that writes an intermediate language token to the intermediate code buffer.

- error.set() and error.status() Routines to register or check for errors.

- Token Token register used to pass the next token from the lexical analyzer to the parser.

- BR Buffer record which contains pointers to the current input stream buffer and intermediate code output buffer. Input buffer switching is mechanized in the parser to facilitate insertion of macro files or symbolic constants into the current stream being processed. A linked list of buffer records dynamically grows and shrinks as needed with BR pointing to the buffer record for the current buffer being processed.

3.3.2 Type Checking

Type checking is accomplished by implementing a simple *type stack*. Whenever the parser emits an object value such as the address of a variable used in an expression or assignment, the type code of that value is pushed to the type stack. Then, when an operation or assignment occurs, embedded type checking routines are executed. For example, suppose the following assignment statement is made where x and y are floating point variables:

x = 4 + y

The intermediate code emitted by the parser for this statement is shown below:

& 329152 i 4 & 329196 @f i2>f + !f

Assume that types FLOAT and INTEGER are coded as **337** and **336,** and the storage addresses of x and y are **329152** and **329196**.

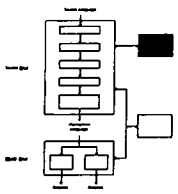
First, the parser recognizes x is a floating point variable being assigned a value and emits the address of x, **& 329152**. The type code of x is then pushed to the type stack, **337**. Next, the parser evaluates the assignment expression 4 + y. The integer number 4 is emitted, **i 4,** and the type code for an integer is pushed to the type stack, **336|337**. The address of y is emitted along with the operator to fetch a floating point value, **& 329196 @f,** then the type code for y is pushed to the type stack, **337|336|337**. The "+" operator pops the two top stack values off the type stack, **337** and **336**, and compares the two type values. At this point, the parser recognizes an implicit cast from integer to float is required to make the two operands the same type. Code is emitted to convert an integer to a float, **i2>f,** and since the result of the operation will yield a float, **337** is pushed onto the type stack, **337|337**. Finally, the plus operator is emitted, **+**. After the assignment expression is evaluated, all that remains is to type check the assignment and emit code to store the expression result at the address of x. The two remaining top stack values are popped off the type stack and compared; the type of the expression and the type of the variable being assigned to, **337** and **337**. Finally, the floating point store operator is emitted, **!f**.

Of course, whenever a type mismatch occurs that cannot be implicitly cast to the correct type, or an operation is undefined for a certain type, an error is generated and parsing terminates.

3.3.3 Intermediate Code Generation

Intermediate code generation is also embedded in the syntax-directed translation scheme. The routine emit() is called whenever an intermediate code token needs to be written to the intermediate code output buffer.

3.4 Symbol Table



The *symbol table* is a data structure used by the translator to store and retrieve records of objects and their associated attributes. Objects are variables, arrays, functions, reserved words, symbolic constants, and types. Object attributes depend on the type of object, but typically include pointers to object lexemes, data types, pointers to data storage locations, parameter attributes and return types for functions, type definition attributes for user defined and base types, etc.

The symbol table dynamically grows during the translation process. Each time an identifier is scanned from the input stream, the symbol table is searched for that identifier. If it's found, attributes are extracted. If not found, it's inserted as a new record into the table and attributes are added as the parsing process continues.

Symbol table routines provide the services to manage the table records. Searching, inserting, extracting, deleting, etc. are some of the processes needed to manage the table records.

3.4.1 Table Structure

The symbol table actually consists of 6 separate linked lists; 1 list for each object type. Each list has a unique record format for the object being stored.

- VariableList variables defined as external in scope
- ArrayList arrays defined as external in scope
- FunctionList all user defined functions
- ReservedList all system reserved words
- SymbolicList all user defined symbolic constants
- TypeList all base and user defined types

Lists can be classified as *global* or *local*.

Global Lists: (6 lists shown above)

VariableList and ArrayList contain variables and arrays defined as external in scope, that is, defined outside of functions. FunctionList, ReservedList, SymbolicList, and TypeList contain functions, reserved words, symbolic constants, and base and user defined types that are always global in scope.

Local Lists:

Variables and arrays that appear in function bodies or parameter lists have records local to that function. Separate variable and array lists are created for each function and pointers to these lists are stored as attributes in the function record. These lists are identical in format to the global lists VariableList and ArrayList.

3.4.2 Scope and Binding Time

Variables and arrays are in general classified as either external or local in scope. To facilitate attribute sharing and binding time, the symbol table refines classification of variables and arrays as follows:

ID_EXTERN	External variable or array
FP_VAL	Function parameter passed as "call by value"
FP_REF	Function parameter passed as "call by reference"
FID_LOCAL	Function local variable or array
FID_EXTERN	Function external variable or array

Although each object has a unique record, some objects share attributes. For example, a function variable declared as **EXTERN** in the body of the function already exists as an external variable that has a record in VariableList. In this case, pointers to the external variable lexeme and storage address will be copied into the function's local variable record.

Binding time refers to that time at which attributes are resolved. In most cases, attributes are resolved at compile time. However, as is the case with reference parameters, some attributes are not resolved until actual runtime.

3.4.3 Record Formats

Record formats for symbol table objects are detailed below.

Variables:

Record Format:

lexeme	*typecode*	*scope*	*address*	*next*

lexeme	ptr to variable name
typecode	integer value representing variable **type**
scope	char value representing variable's scope; **ID_EXTERN, FP_VAL, FP_REF, FID_LOCAL, FID_EXTERN**
address	ptr to first element in variable storage block
next	ptr to next variable record in table

Scope and Attribute Creation: Bold means shared attribute; capital letters denote runtime resolution, lowercase denote compile time resolution.

ID_EXTERN	*lexeme*	*typecode*	*scope*	*address*	*next*
FP_VAL	*lexeme*	*typecode*	*scope*	*address*	*next*
FP_REF	*lexeme*	*typecode*	*scope*	***ADDRESS***	*next*
FID_LOCAL	*lexeme*	*typecode*	*scope*	*address*	*next*
FID_EXTERN	*lexeme*	*typecode*	*scope*	***address***	*next*

Arrays:

Record Format:

lexeme	*typecode*	*scope*	*address*	*size*	*width*	*dimentotal*	*term*	*indexlist*	*next*

lexeme	ptr to array name
typecode	integer value representing array element **type**
scope	char value representing array's scope; **ID_EXTERN, FP_VAL, FP_REF, FID_LOCAL, FID_EXTERN**
address	ptr to first element in array storage block
size	total number of elements in array
width	storage field width of a single array element (in bytes)
dimentotal	number of array dimensions
term	a constant term used in calculating the storage offset of array elements
indexlist	ptr to array dimension index list
next	ptr to next array record in table

Local Record Format: *indexlist*

1st dimension	*lower*	*upper*	*size*
2nd	*lower*	*upper*	*size*
3rd	*lower*	*upper*	*size*
:	:	:	:

lower	lower array bound of dimension
upper	upper array bound of dimension
size	number of elements in dimension (upper-lower+1)

Scope and Attribute Creation: Bold means shared attribute; capital letters denote runtime resolution, lowercase denote compile time resolution. Shaded means not applicable; arrays cannot be passed by value

ID_EXTERN	*lexeme*	*typecode*	*scope*	*address*	*size*	*width*	*dimentotal*	*term*	*indexlist*	*next*
FP_VAL										
FP_REF	*lexeme*	*typecode*	*scope*	***ADDRESS***	*SIZE*	*WIDTH*	*DIMENTOTAL*	*TERM*	***INDEXLIST***	*next*
FID_LOCAL	*lexeme*	*typecode*	*scope*	*address*	*size*	*width*	*dimentotal*	*term*	*indexlist*	*next*
FID_EXTERN	***lexeme***	*typecode*	*scope*	***address***	*size*	*width*	*dimentotal*	*term*	***indexlist***	*next*

Reserved Keywords:

Record Format:

lexeme	*keyword code*	*next*

lexeme	ptr to reserved keyword name
keyword code	unique integer value > 255 assigned to reserved keyword
next	ptr to next reserved keyword record in table

Symbolic Constants:

Record Format:

lexeme	*string*	*next*

lexeme	ptr to symbolic constant name
string	ptr to replacement string
next	ptr to next symbolic constant record in table

Functions:

Record Format:

lexeme	*typecode*	*address*	*numparams*	*plist*	*vlist*	*alist*	*next*

lexeme	ptr to function lexeme
typecode	integer value representing **type** of function return value
address	ptr to first element in function return value storage block
numparams	integer value representing number of function parameters
plist	pointer to function parameter list
vlist	pointer to function variable list
alist	pointer to function array list
next	ptr to next function record in table

Local Record Format: *plist*

lexeme	*typecode*	*scope*	*recordtype*	*next*

lexeme	ptr to parameter lexeme
typecode	integer value representing **type** of parameter
scope	char value representing parameter's scope; **ID_EXTERN, FP_VAL, FP_REF, FID_LOCAL, FID_EXTERN**
recordtype	integer value representing kind of parameter; **VARIABLE, ARRAY**
next	ptr to next parameter record in parameter list

Local Record Format: *vlist*

The *vlist* is a linked list of variable records local to the function body. Function variable parameters are also stored at the head of this list. Same record format as Variables.

Local Record Format: *alist*

The *alist* is a linked list of array records local to the function body. Function array parameters are also stored at the head of this list. Same record format as Arrays.

Types:

Record Format:

lexeme	*typecode*	*total units*	*total bytes*	*segment offset list*	*member list*	*next*

lexeme	ptr to type name
typecode	unique integer value > 255 assigned to type
total units	total number of base type storage elements required for type
total bytes	total number of storage bytes required for type
segment offset list	storage offset index for elements
member list	ptr to type member list, 0 if base type
next	ptr to next type record in table

Local Record Format: *member list*

lexeme	*typeptr*	*segment start*	*segment end*	*next*

lexeme	ptr to member name
typeptr	ptr to member's type record in type table; e.g. member **POINT R1** would point to type record for type **POINT**
segment start	member's starting unit offset relative to type's total unit list
segment end	member's ending unit offset relative to type's total unit list
next	ptr to next member record in member list

Local Record Format: *segment offset list*

typecode	*byte offset*
typecode	*byte offset*
typecode	*byte offset*
:	:
0	*next free byte*

typecode	integer value representing member **type**
byte offset	relative offset byte of member's data storage location
next free byte	next free byte relative to member's data storage location

Example: Suppose a user defined type contains as members an int (type 336), float (type 337), float, and int. It's segment offset list would be as follows:

336	*0*
337	*4*
337	*12*
336	*20*
0	*24*

3.4.4 Service Routines

The major service routines for managing symbol table objects are listed below with a brief description of what each does.

Variables:

InsertVariable( )	Insert a new variable record into the symbol table.

Arrays:

InsertArray( )	Insert a new array record into the symbol table.
CalcArrayOffset()	Calculate the byte offset of an array element's storage position with respect to the starting address of the array's storage block

Reserved Keywords:

InsertReserved( )	Insert a new reserved keyword record into the symbol table. This routine is called once at startup to initialize the symbol table with reserved keywords.

Symbolic Constants:

InsertSymbolic( )	Insert a new symbolic constant record into the symbol table.

Functions:

InsertFunction()	Insert a new function record into the symbol table.
LookupParameter()	Search function's parameter list for a parameter.

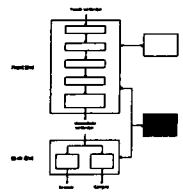
Types:

InsertBasetype( )	Insert base types into type table at startup (FLOAT, INTEGER, etc.)
InsertTypeDef( )	Insert a new user defined type record into the symbol table.
InsertMember( )	Insert a member into the member list of the current type under construction.
LookupType( )	Search type table for a type.
LookupMember( )	Search a type's member list for a member record.
BuildOffsetList( )	Builds a storage index array for accessing type member storage locations.

General Routines:

SetScope()	Routine that sets the current VariableList and ArrayList to either global or local to a function.
SetState()	Records the current state of the symbol table lists. Used to restore the symbol table in the case where a system error occurs.
RestoreState()	Routine that restores the symbol table lists to the condition recorded with a call to SetState().
Delete()	Routine to delete an object record from the symbol table.
MakeStorage()	Routine to allocate storage for a new object.
Lookup()	General lookup routine for any symbol table object.

3.5 Error Module



Error management is integral to all phases of the translation process. Runtime error support is also included for the detection of floating point errors and array bounds exceptions.

Typical error management services include:

- Registration, detection, and display of errors during translation and runtime phases
- Comprehensive error messages
- Display of source line where error occurred and if applicable, line number and filename
- Color highlighting of token or expression that generated the error

3.5.1 Interface Routines and Record Format

Basically, the error module consists of a linked list of error records that are registered during the execution of the translator and three interface routines to set, check, and display/clear the error records. Interface routines and record format are described below:

Interface routines:

error.status()	Check error module to see if an error(s) has occurred.
error.set()	Call error module to register an error.
error.show()	Call error module to display errors and reset the error register.

Record Format:

number	*type*	*code*	*msg*	*text*	*lineno*	*next*

number	number of error with respect to current error list
type	general error classification; LEXICAL, PARSER, SYMTABLE, COMP, IOERR, RUNTIME, STACKS, FPE.
code	integer code used to assist in highlighting error messages
msg	ptr to error message
text	ptr to source code line containing error
lineno	source code line number where error occurs
next	ptr to next error record in the current error list

3.5.2 Error Management Rules

Error management and control is methodically handled by adhering to the following set of rules:

Rule 1. Each function in the source code must have as its first statement a call to error.status() to check if an error has been registered. If so, program control must be returned to the calling routine without executing any code in the current function.

Rule 2. If an error can occur within the body of a function, code must be included to call error.set() to register the error and then return program control to the calling routine.

Rule 3. If a function calls another function, or contiguous sequence of functions, the next statement must be either:

a.) A call to error.status() to check if the called function(s) registered an error, and if so, program control must be returned to the calling routine without executing any more code in the body of the current function.

b.) A return of program control to the calling routine.

Rule 4. A single controlling executive program or function must exist where return will eventually come to and code will be executed to display the registered errors and reset the translator.

Rule 1. basically shuts down all functions once an error has been registered. Source code may still continue to call functions as is the case when a function calls several functions sequentially, but they don't do anything except return.

Rule 2. requires every function to manage and register any errors its responsible for and return program control to the calling routine in this event.

Rule 3. requires every function that calls another function to perform some task to check if that called function generated an error, and if so, stop and return.

Rule 4. states that there must exist some top level function where program control must always return to so that registered errors can be displayed and the translator/application can be reset.

3.5.3 Example

A typical code fragment is shown below to illustrate embedding of error management in the source code.

```
void open_statement()
{
 if(error.status()) return;
 match(OPEN);channel();match(',');mode();match(',');filename();emit("open");
}

void channel()
{
 if(error.status()) return;
 switch(Token.Type)
 {
  case('#'): match('#'); Expr(); break;
  default:
   error.set(0,PARSER,"Channel number must be prefixed with # symbol");
   return;
 }
 if(error.status()) return;
 if(popt()!=INTEGER)
 {
  error.set(0,PARSER,"Channel number must be an INTEGER expr in range 1 to 255");
  return;
 }
}
```

Suppose, for example, a call is made to the function open_statement() to open a file on disk. The first action of the function open_statement() is to make a call to the error module to see if an error has been registered prior to it being called (Rule 1.) Next, a contiguous sequence of function calls are made to the functions match(), channel(), match(), mode(), match(), filename(), and emit(). Finally, open_statement() returns program control to whatever function called it (Rule 3b.) Since no error can be generated in the body of open_statement(), Rule 2. does not apply in this case. If an error has been generated at some point executing the open_statement(), program control will return eventually to a single controlling portion of the program where the error(s) can be displayed and the system reset (Rule 4.)

Now examine the function channel() which is called by open_statement().The first action of the function channel() is to make a call to the error module to see if an error has been registered prior to it being called (Rule 1.) An error at this point would exist only if the preceding call to the function match() in open_statement() resulted in an error being registered by itself or some function it called. If an error exists, program control is returned to open_statement(). If no error exists, the next step is to process the switch block in the body of channel().

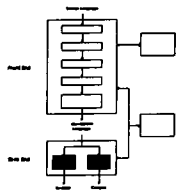
If case = default, the function channel() has determined an error must be generated, a call is made to error.set() to register the error, and program control is then returned to open_statement() (Rule 2.)

If case = '#', a contiguous sequence of function calls are made to the functions match() and Expr(). Next, a call is made to the error module to see if an error has been registered by one of the functions called (Rule 3a.) If an error has occurred, program control is returned to open_statement(). If no error has been registered, the conditional if block is tested for popt()!=INTEGER. If true, the function channel() has determined an error must be generated, a call is made to error.set() to register the error, and program control is then returned to open_statement() (Rule 2.) If false, program control returns to open_statement() with no registered errors.

3.5.4 Design Note - Multiple Error Registration

Currently, translation stops at the occurrence of the first error detected so a linked list of records is not necessary. However, future translator development would desirably incorporate some means of error correction or method of error recovery to continue compilation. This would require a mechanism to register multiple errors and was the main reason for structuring the error register with this capability now. However, as it stands, multiple errors can occur if the source code fails to return properly after the first occurrence of an error. Therefore, the registration of multiple errors currently serves as a debugging tool for program development.

3.6 Threaded Interpreter/Compiler



The back end interpreter and code generator is a *threaded code interpreter*. As stated in the theory section of this paper, a threaded code interpreter is itself a translator that incorporates a threaded code generator and software interpreter. Threaded code is a fully analyzed internal form of instructions comprised of addresses that point to either primitives, secondaries, or literals.

Using a core set of primitives, stacks and registers, and maintaining a threaded code keyword dictionary, the interpreter mechanizes two modes of operation; a *compile mode* where secondaries are created and an *execution mode* where threaded code is interpreted and executed.

The major components of the threaded interpreter/compiler are:

- Primitives and secondaries.
- TIL keyword dictionary.
- Stacks and Registers.
- Supporting machine code routines.
- Outer interpreter controlling executive.
- Inner interpreter.

A complete description of a TIL is given in the theory section of this paper. Figure 3.7 shows the main elements of the TIL.

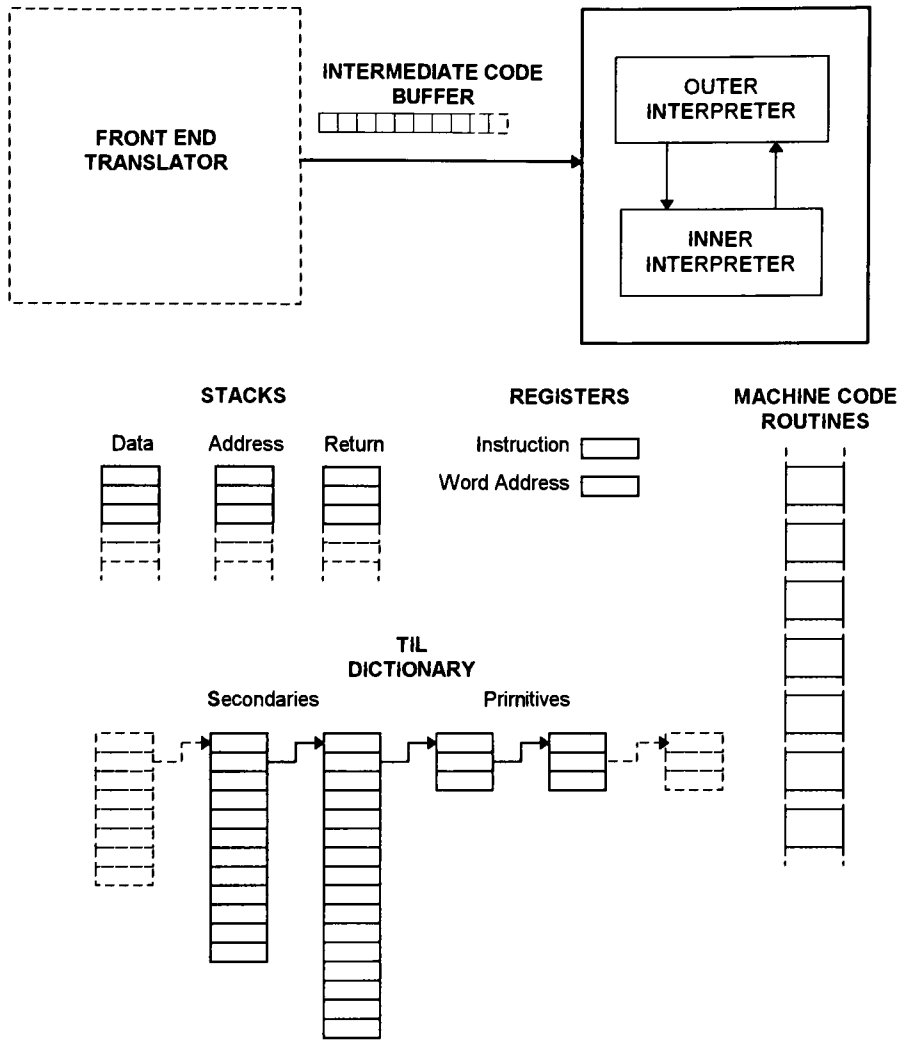


Figure 3.7 Threaded code interpreter/compiler.

3.6.1 Primitives and Secondaries

Primitives are the core keywords of the language. For each primitive, there exists a corresponding machine code routine to perform the action the primitive was designed to do. Secondaries group pointers to other secondaries and primitives and structure lists of threaded code instructions. Secondaries mechanize language extension, similar to functions, while primitives form the base language for the system. Shown below are the record formats for primitives and secondaries.

Primitive Record Format:

lexeme	*link*	*code address*

lexeme ptr to keyword name
link ptr to next keyword record in TIL dictionary
code address ptr to primitive's machine code routine

Secondary Record Format:

lexeme	*link*	*code address*	*word address #1*	*word address #2*	...	*SEMI address*

lexeme ptr to keyword name
link ptr to next keyword record in TIL dictionary
code address ptr to machine code routine COLON
word address ptr to *code address* section of primitive or secondary keyword record
SEMI *address* ptr to *code address* section of primitive keyword record SEMI

A descriptive listing of all system primitives and secondaries is given in Table 3.5 thru Table 3.9. For short routines, the actual actions of the routine are described. For longer routines, a general description is given. See the source code listings in the Appendix for details about a particular routine.

KEYWORD	DESCRIPTION
inner_til_exit	routine to set inner interpreter exit flag TRUE
semi	inner interpreter routine SEMI
float_lh	floating point literal handler routine
int_lh	integer literal handler routine
vptr_lh	void pointer literal handler routine
adr_lh	address literal handler routine
str_lh	string literal handler routine
_swtch	runtime routine to pop result of switch floating point expression from DS and store in temporary register
_SWTCH	runtime routine to pop result of switch integer expression from DS and store in temporary register
_if	runtime routine to evaluate result of conditional expression and branch to appropriate location in threaded code list
_ifeq	runtime routine to evaluate floating point results of case and switch expression and branch to appropriate location in threaded code list
_IFEQ	runtime routine to evaluate integer results of case and switch expression and branch to appropriate location in threaded code list
_jmp	runtime routine to mechanize jump to new location in threaded code list
rtnerr	routine to register error when function structure causes a return with no value

Table 3.5 TIL headerless primitives.

KEYWORD	DESCRIPTION
inner_til_exit_adr	threaded code address of headerless primitive inner_til_exit which is used to set inner interpreter exit flag TRUE

Table 3.6 TIL headerless secondaries.

KEYWORD	DESCRIPTION
dump	routine to toggle threaded code listing on or off for debugging
rtn	routine to mechanize a RETURN from a function
rtn?	routine to check if a secondary properly terminates with a call to SEMI
&	set stack to address mode
i	set stack to integer mode
ii	set stack to immediate integer mode
f	set stack to floating point double mode
&s	set stack to string pointer mode
&v	set stack to void pointer mode

Table 3.7 TIL immediate vocabulary primitives.

KEYWORD	DESCRIPTION
;	routine to terminate a secondary and insert call to SEMI
if	routine used in constructing an IF block
elseif	routine used in constructing an IF block
endif	routine used in constructing an IF block
jmp	routine to mechanize jump to new location in threaded code list
while	routine used in constructing a WHILE block
wend	routine used in constructing a WHILE block
repeat	routine used in constructing a REPEAT block
until	routine used in constructing a REPEAT block
for	routine used in constructing a FOR block
for1	routine used in constructing a FOR block
for2	routine used in constructing a FOR block
next	routine used in constructing a FOR block
break	routine to mechanize a BREAK statement
setbrk	routine to mechanize a BREAK statement
swtch	routine used in constructing a SWITCH block
SWTCH	routine used in constructing a SWITCH block
case1	routine used in constructing a SWITCH block
CASE1	routine used in constructing a SWITCH block
case2	routine used in constructing a SWITCH block
case3	routine used in constructing a SWITCH block
CASE3	routine used in constructing a SWITCH block
send	routine used in constructing a SWITCH block
SEND	routine used in constructing a SWITCH block
default	routine used in constructing a SWITCH block
cjmp	routine used in constructing a SWITCH block

Table 3.8 TIL compiler vocabulary primitives.

KEYWORD	DESCRIPTION
:	inner interpreter routine COLON
list	routine to list TIL keywords to screen
i>f	convert top DS entry from integer to floating point
i2>f	convert 2nd DS entry from integer to floating point
@f	pop address from AS, fetch floating point number stored at address, push to DS
@f+	pop address from AS and offset from DS, fetch floating point number stored at address+offset, push to DS
@F	copy address from AS, fetch floating point number stored at address, push to DS
@F+	copy address from AS and pop offset from DS, fetch floating point number stored at address+offset, push to DS
!f	pop address from AS and floating point number from DS, store number at address
!f+	pop address from AS and offset from DS, pop floating point number from DS, store number at address+offset
!F	copy address from AS, pop floating point number from DS, store number at address
!F+	copy address from AS, pop offset from DS, pop floating point number from DS, store number at address+offset
@i	pop address from AS, fetch integer number stored at address, push to DS
@i+	pop address from AS and offset from DS, fetch integer number stored at address+offset, push to DS
@I	copy address from AS, fetch integer number stored at address, push to DS
@I+	copy address from AS and pop offset from DS, fetch integer number stored at address+offset, push to DS
!i	pop address from AS and integer number from DS, store number at address
!i+	pop address from AS and offset from DS, pop integer number from DS, store number at address+offset
!I	copy address from AS, pop integer number from DS, store number at address
!I+	copy address from AS, pop offset from DS, pop integer number from DS, store number at address+offset
@&s	pop address from AS, fetch string pointer stored at address, push to DS
@&s+	pop address from AS and offset from DS, fetch string pointer stored at address+offset, push to DS
@&S	copy address from AS, fetch string pointer stored at address, push to DS
@&S+	copy address from AS and pop offset from DS, fetch string pointer stored at address+offset, push to DS
!&s	pop address from AS and string pointer from DS, store string pointer at address
!&s+	pop address from AS and offset from DS, pop string pointer from DS, store string pointer at address+offset
!&S	copy address from AS, pop string pointer from DS, store string pointer at address
!&S+	copy address from AS, pop offset from DS, pop string pointer from DS, store string pointer at address+offset

Table 3.9 TIL core vocabulary primitives.

KEYWORD	DESCRIPTION
!&v	pop address from AS and void pointer from DS, store void pointer at address
@&va	pop address from DS, fetch void pointer at address, push void pointer to AS
@&vd	pop address from DS, fetch void pointer at address, push void pointer to DS
aof	routine to calculate address offset of an array element
aiof	routine to calculate address offset of an array element
of+	pop address from AS and offset from DS, push address+offset to AS
OF+	copy address from AS, pop offset from DS, push address+offset to AS
acpy	pop target array record address from AS and source array record address from DS, copy array record entries from source to target
*	pop 2 top stack floating point entries from DS, floating point multiply, push result to DS
/	pop 2 top stack floating point entries from DS, floating point divide, push result to DS
+	pop 2 top stack floating point entries from DS, floating point add, push result to DS
-	pop 2 top stack floating point entries from DS, floating point subtract, push result to DS
i*	pop 2 top stack integer entries from DS, integer multiply, push result to DS
i+	pop 2 top stack integer entries from DS, integer add, push result to DS
i-	pop 2 top stack integer entries from DS, integer subtract, push result to DS
s+	pop 2 top stack string pointer entries from DS, fetch strings at addresses, concatenate, push pointer to concatenated string to DS
neg	pop floating point number from DS, negate, push result to DS
NEG	pop integer number from DS, negate, push result to DS
x<>0	pop floating point number from DS, if not equal to 0, push integer 1 to DS, else push integer 0 to DS
!x	pop floating point number from DS, if not equal to 0, push integer 0 to DS, else push integer 1 to DS
!X	pop integer number from DS, if not equal to 0, push integer 0 to DS, else push integer 1 to DS
y<x	pop 1st and 2nd floating point numbers from DS, if 2nd is less than 1st, push integer 1 to DS, else push integer 0 to DS
Y<X	pop 1st and 2nd integer numbers from DS, if 2nd is less than 1st, push integer 1 to DS, else push integer 0 to DS
y<=x	pop 1st and 2nd floating point numbers from DS, if 2nd is less than or equal to 1st, push integer 1 to DS, else push integer 0 to DS
Y<=X	pop 1st and 2nd integer numbers from DS, if 2nd is less than or equal to 1st, push integer 1 to DS, else push integer 0 to DS
y>x	pop 1st and 2nd floating point numbers from DS, if 2nd is greater than 1st, push integer 1 to DS, else push integer 0 to DS
Y>X	pop 1st and 2nd integer numbers from DS, if 2nd is greater than 1st, push integer 1 to DS, else push integer 0 to DS
y>=x	pop 1st and 2nd floating point numbers from DS, if 2nd is greater than or equal to 1st, push integer 1 to DS, else push integer 0 to DS

Table 3.9 (cont.) TIL core vocabulary primitives.

KEYWORD	DESCRIPTION
Y>=X	pop 1st and 2nd integer numbers from DS, if 2nd is greater than or equal to 1st, push integer 1 to DS, else push integer 0 to DS
y=x	pop 1st and 2nd floating point numbers from DS, if 2nd is equal to 1st, push integer 1 to DS, else push integer 0 to DS
Y=X	pop 1st and 2nd integer numbers from DS, if 2nd is equal to 1st, push integer 1 to DS, else push integer 0 to DS
y<>x	pop 1st and 2nd floating point numbers from DS, if 2nd is not equal to 1st, push integer 1 to DS, else push integer 0 to DS
Y<>X	pop 1st and 2nd integer numbers from DS, if 2nd is not equal to 1st, push integer 1 to DS, else push integer 0 to DS
y&x	pop 1st and 2nd floating point numbers from DS, if 2nd AND 1st, push integer 1 to DS, else push integer 0 to DS
Y&X	pop 1st and 2nd integer numbers from DS, if 2nd AND 1st, push integer 1 to DS, else push integer 0 to DS
y|x	pop 1st and 2nd floating point numbers from DS, if 2nd OR 1st, push integer 1 to DS, else push integer 0 to DS
Y|X	pop 1st and 2nd integer numbers from DS, if 2nd OR 1st, push integer 1 to DS, else push integer 0 to DS
sin	pop floating point number from DS, calculate sine, push result to DS
cos	pop floating point number from DS, calculate cosine, push result to DS
tan	pop floating point number from DS, calculate tangent, push result to DS
asin	pop floating point number from DS, calculate arcsine, push result to DS
acos	pop floating point number from DS, calculate arccosine, push result to DS
atan	pop floating point number from DS, calculate arctangent, push result to DS
sinh	pop floating point number from DS, calculate hyperbolic sine, push result to DS
cosh	pop floating point number from DS, calculate hyperbolic cosine, push result to DS
tanh	pop floating point number from DS, calculate hyperbolic tangent, push result to DS
exp	pop floating point number from DS, calculate $e^{(number)}$, push result to DS
ln	pop floating point number from DS, calculate natural log, push result to DS
log	pop floating point number from DS, calculate base 10 log, push result to DS
sqrt	pop floating point number from DS, calculate square root, push result to DS
abs	pop floating point number from DS, calculate absolute value, push result to DS
y^x	pop 1st and 2nd floating point numbers from DS, raise 2nd to power 1st, push result to DS
mem	routine to display TIL dictionary memory usage
prtf	pop floating point number from DS, print to screen
prti	pop integer number from DS, print to screen
prts	pop string pointer from DS, print string to screen
prt	print newline character to screen

Table 3.9 (cont.) TIL core vocabulary primitives.

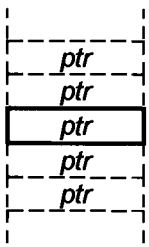
KEYWORD	DESCRIPTION
fprtf	pop floating point number from DS, print to file
fprti	pop integer number from DS, print to file
fprts	pop string pointer from DS, print string to file
fprt	print newline character to file
cprt	routine to set new file channel
open	routine to open new file
close	routine to close a file
closea	routine to close all files
input	routine to extract line of input from keyboard or file
inpf	routine to process floating point expression
inpi	routine to process integer expression
inps	routine to process string expression
ftoa	routine to convert floating point number to a string
trans	routine to call the translator and process a source language stream
sys	routine to shell out of translator to execute a program
del	routine to delete an object
local	routine to set translator symbol table scope local to a function
global	routine to set translator symbol table scope to global

Table 3.9 (cont.) TIL core vocabulary primitives.

3.6.2 TIL Keyword Dictionary

The TIL dictionary is simply an array of pointers forming a contiguous block of memory allocated at system startup. Primitives and secondaries, as well as pointers to literals, are structured and stored sequentially in this memory block. Each entry in the dictionary is a generic pointer as shown below.

TIL Dictionary Format:



ptr generic pointer to a literal, machine code routine, or dictionary entry

Vocabularies segment the dictionary into different areas. The TIL dictionary is structured with the following vocabularies:

HEADERLESS	Bottom of dictionary space used to store headerless system primitives and secondaries.
COMPILER	System primitives that are executed during compile mode, invalid in execute mode.
IMMEDIATE	System primitives that are always executed regardless of mode.
CORE	Bulk of system primitives; compile in compile mode, execute in execute mode.
USER	Free space for insertion of user created secondaries.

At startup, all system primitives and secondaries are created and inserted into the dictionary vocabularies HEADERLESS, COMPILER, IMMEDIATE, and CORE. The remaining dictionary space is allocated to USER. At present, the dictionary memory block is fixed in size at system startup. Once the dictionary space becomes filled, no more secondaries can be compiled into the system.

3.6.3 Stacks and Registers

The threaded code interpreter uses the following stacks and registers:

DS	Data Stack; LIFO stack used to store numbers
AS	Address Stack; LIFO stack used to store addresses
RS	Return Stack; LIFO stack used to store return addresses when a secondary calls another secondary or a primitive.
I_reg	Instruction Register; used to store address of next threaded code instruction in current secondary being processed.
WA_reg	Word Address Register; used to store the word address of the current keyword or the address of the code body section of the current keyword.

Registers are simply generic pointers. Stacks are fixed length arrays of generic pointers allocated at system startup.

Note that no code address register, CA_reg, is implemented as was described in the theory section. Since speed is critical to the operation of the inner interpreter, this register was eliminated. During execution, WA_reg points to the entry containing the code address of the executable routine so there's no need to move the address to another register and then call the routine.

Also note the inclusion of an additional stack, the Address Stack. This stack was implemented to store addresses separate from data to make the design of the interpreter easier to implement.

3.6.4 Outer Interpreter

The outer interpreter serves as the controlling executive for the interpreter. The outer interpreter performs the following tasks:

- Token extraction and classification; number, address, or keyword.
- Compilation of threaded code.
- Management of system flags and registers.

A flow chart of the basic operation of the outer interpreter is shown in Figure 3.8.

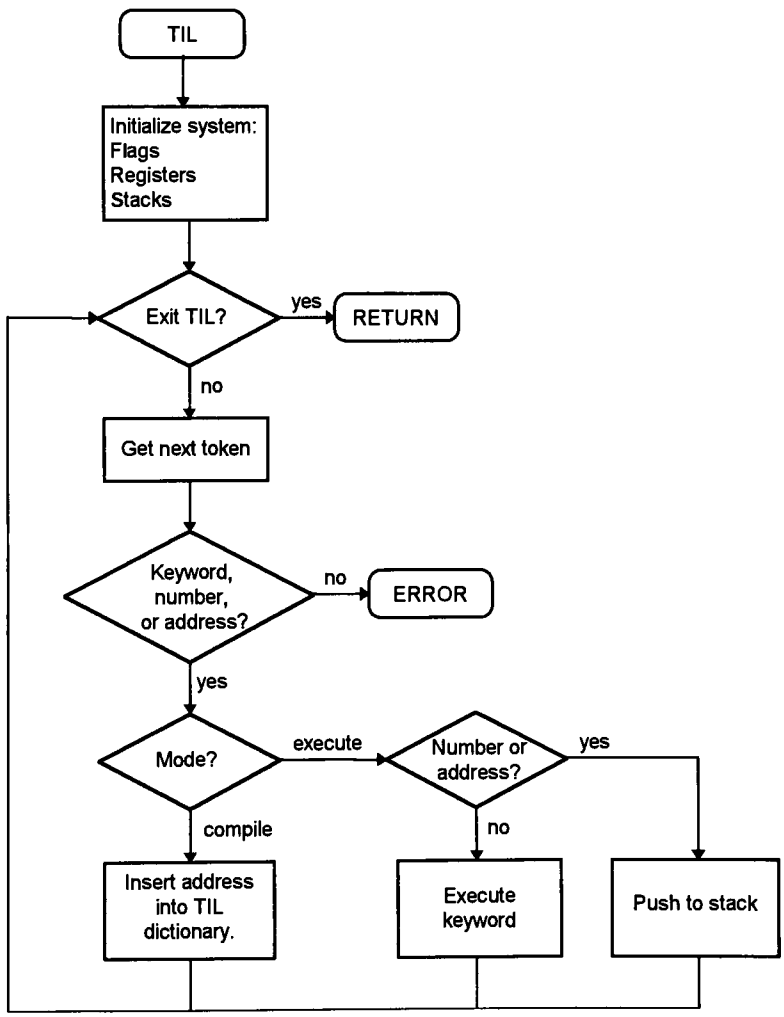


Figure 3.8 Flowchart of basic outer interpreter operation.

Several routines support the operation of the outer interpreter:

til() The main controlling executive.

create() Routine to create header for new secondary.

token() Routine to extract next token from the intermediate code buffer.

search() Routine to search TIL dictionary for keyword.

execute() Routine to execute current keyword or compile into current secondary under construction.

number() Routine to recognize token as a number or address, push to stack or compile into current secondary under construction.

3.6.5 Inner Interpreter

The inner interpreter is a small loop that mechanizes the execution of threaded code. Shown below is the actual code comprising the inner interpreter:

```
reg_WA=KEYWORD;          // initialize reg_WA with address of keyword code body

reg_I=I_start_adr;       // initialize reg_I to exit inner til instruction

while(FLAG.INNER)        // This while() block is the "inner interpreter".
{                        // funcp() executes the current keyword code
 reg_WA->funcp();        // body. Loop until last code body executed sets
 reg_WA=reg_I++->dictp;  // FLAG.INNER = OFF.
}
```

3.7 Source Language Specification

A complete four part specification of the context-free grammar used in the translator is presented in the pages that follow. Recall from the theory section, a context-free grammar is composed of terminals, nonterminals, productions, and a starting nonterminal.

Notation conventions used in specifying the grammar are as follows:

⟨ ⟩	Nonterminals; e.g., ⟨*object*⟩ ⟨*more_objects*⟩.
ε	Empty set; allows substitution of "null sequence" for a nonterminal.
[ ]	Optional; e.g., ["**&**"] ⟨*id*⟩ means ampersand is optional, "**&**"⟨*id*⟩ or ⟨*id*⟩ are acceptable.
{ }	Repeating; e.g., { ⟨*digit*⟩ } means ⟨*digit*⟩⟨*digit*⟩⟨*digit*⟩... to whatever length required.
|	Alternative choice; e.g., "**+**" | "**-**" means "**+**" or "**-**" is acceptable.
→	Production symbol; single nonterminal to left of symbol has as a production a finite sequence of terminals and/or nonterminals given to right of symbol. If alternative productions for a nonterminal exist, each is grouped with the nonterminal and preceded by the → symbol.

Boldface words in quotes represent individual terminal symbols; e.g., "**LET**", "**+**", "**TYPEDEF**", etc.

Boldface italicized words in quotes represent a general set of terminal symbols from which one will be substituted; e.g., "***type***" could be "**FLOAT**", "**INTEGER**", "***unsigned real***" could be "**1.25**", etc.

Boldface nonterminals are used just to emphasize the major nonterminals, they have no significance.

1. A finite, nonempty set of symbols called *terminals*. See Table 3.10.

ε	.	,	;	:
&	(	)	[	]
{	}	=	<>	<
<=	>	>=	+	-
*	/	^	OR	AND
NOT	LET	TYPEDEF	DEFINE	END_DEFINE
RETURN	FOR	NEXT	GLOBAL	LOCAL
IF	ELSEIF	ELSE	ENDIF	EXTERN
SWITCH	CASE	DEFAULT	ENDSWITCH	BREAK
WHILE	ENDWHILE	REPEAT	UNTIL	TRANSLATE
FLOAT	INTEGER	STRING	INPUT	PRINT
OPEN	CLOSE	LOAD	FTOA	DELETE
SYMBOL	SYSTEM	SIN	COS	TAN
ASIN	ACOS	ATAN	SINH	COSH
TANH	ABS	SQRT	LOG	LN
EXP	*identifier*	*string*	*unsigned real*	*unsigned integer*
user defined type				

Table 3.10 Terminal set.

2. A finite, nonempty set of *nonterminals*. See Table 3.11.

⟨**command_stream**⟩	⟨**input_statement**⟩	⟨**external_declaration_statement**⟩
⟨more_objects⟩	⟨prompt⟩	⟨**conditional_statement**⟩
⟨**object**⟩	⟨channel⟩	⟨if_statement⟩
⟨**command_statement**⟩	⟨var_list⟩	⟨switch_statement⟩
⟨**declaration_statement**⟩	⟨more_vars⟩	⟨**iterative_statement**⟩
⟨new_identifier_list⟩	⟨var⟩	⟨for_statement⟩
⟨more_new_identifiers⟩	⟨**print_statement**⟩	⟨while_statement⟩
⟨new_identifier⟩	⟨expression_list⟩	⟨repeat_statement⟩
⟨dimension_list⟩	⟨more_expressions⟩	⟨**return_statement**⟩
⟨more_dimensions⟩	⟨**open_statement**⟩	⟨**break_statement**⟩
⟨dimension⟩	⟨mode⟩	⟨**expression**⟩
⟨**assignment_statement**⟩	⟨filename⟩	⟨moreterm1s⟩
⟨assignment_list⟩	⟨**close_statement**⟩	⟨term1⟩
⟨more_assignments⟩	⟨**translate_statement**⟩	⟨moreterm2s⟩
⟨assignment⟩	⟨**delete_statement**⟩	⟨term2⟩
⟨**identifier**⟩	⟨delete_list⟩	⟨moreterm3s⟩
⟨indice_list⟩	⟨more_delete_expressions⟩	⟨term3⟩
⟨more_indices⟩	⟨delete_expression⟩	⟨moreterm4s⟩
⟨indice⟩	⟨**symbolic_statement**⟩	⟨term4⟩
⟨member_list⟩	⟨**system_statement**⟩	⟨moreterm5s⟩
⟨more_members⟩	⟨**local_statement**⟩	⟨term5⟩
⟨member⟩	⟨**global_statement**⟩	⟨moreterm6s⟩
⟨**typedef_statement**⟩	⟨**function_definition**⟩	⟨term6⟩
⟨type_name⟩	⟨return_type⟩	⟨term7⟩
⟨new_member_list⟩	⟨new_function_name⟩	⟨term8⟩
⟨more_new_members⟩	⟨param_list⟩	⟨**type**⟩
⟨new_member⟩	⟨more_params⟩	⟨**id**⟩
⟨new_member_name⟩	⟨param⟩	⟨**number**⟩
⟨**function_statement**⟩	⟨param_type⟩	⟨**sliteral**⟩
⟨function_name⟩	⟨param_name⟩	
⟨parameter_list⟩	⟨**statement_list**⟩	
⟨more_parameters⟩	⟨more_statements⟩	
⟨parameter⟩	⟨statement⟩	

Table 3.11 Nonterminal set.

3. A finite, nonempty set of rules called *productions*. Productions for the entire language are listed below.

⟨***command_stream***⟩ → ⟨*object*⟩ ⟨*more_objects*⟩
⟨*more_objects*⟩ → ⟨*object*⟩ ⟨*more_objects*⟩
→ ε
⟨*object*⟩ → ⟨*function_definition*⟩
→ ⟨*command_statement*⟩
→ ε

⟨***command_statement***⟩ → ⟨*declaration_statement*⟩
→ ⟨*assignment_statement*⟩
→ ⟨*typedef_statement*⟩
→ ⟨*function_statement*⟩
→ ⟨*input_statement*⟩
→ ⟨*print_statement*⟩
→ ⟨*open_statement*⟩
→ ⟨*close_statement*⟩
→ ⟨*translate_statement*⟩
→ ⟨*delete_statement*⟩
→ ⟨*symbolic_statement*⟩
→ ⟨*system_statement*⟩
→ ⟨*local_statement*⟩
→ ⟨*global_statement*⟩

⟨***declaration_statement***⟩ → ⟨*type*⟩ ⟨*new_identifier_list*⟩
⟨*new_identifier_list*⟩ → ⟨*new_identifier*⟩ ⟨*more_new_identifiers*⟩
⟨*more_new_identifiers*⟩ → "," ⟨*new_identifier*⟩ ⟨*more_new_identifiers*⟩
→ ε
⟨*new_identifier*⟩ → ⟨*id*⟩
→ ⟨*id*⟩ "["⟨*dimension_list*⟩"]"
⟨*dimension_list*⟩ → ⟨*dimension*⟩ ⟨*more_dimensions*⟩
⟨*more_dimensions*⟩ → "," ⟨*dimension*⟩ ⟨*more_dimensions*⟩
→ ε
⟨*dimension*⟩ → [[⟨*sign*⟩] ⟨*integer*⟩ ":"] [⟨*sign*⟩] ⟨*integer*⟩

⟨***assignment_statement***⟩ → ["LET"] ⟨*assignment_list*⟩

⟨***assignment_list***⟩ → ⟨*assignment*⟩ ⟨*more_assignments*⟩
⟨*more_assignments*⟩ → "," ⟨*assignment*⟩ ⟨*more_assignments*⟩
→ ε
⟨*assignment*⟩ → ⟨*identifier*⟩ "=" ⟨*expression*⟩

⟨***identifier***⟩ → ⟨*id*⟩ ⟨*indice_list*⟩ ⟨*member_list*⟩
⟨*indice_list*⟩ → "[" ⟨*indice*⟩ ⟨*more_indices*⟩ "]"
→ ε
⟨*more_indices*⟩ → "," ⟨*indice*⟩ ⟨*more_indices*⟩
→ ε
⟨*indice*⟩ → ⟨*expression*⟩
⟨*member_list*⟩ → "." ⟨*member*⟩ ⟨*more_members*⟩
→ ε
⟨*more_members*⟩ → "." ⟨*member*⟩ ⟨*more_members*⟩
→ ε
⟨*member*⟩ → ⟨*id*⟩

⟨***typedef_statement***⟩ → "TYPEDEF" ⟨*type_name*⟩ "{" ⟨*new_member_list*⟩ "}"
⟨*type_name*⟩ → ⟨*id*⟩
⟨*new_member_list*⟩ → ⟨*new_member*⟩ ⟨*more_new_members*⟩
⟨*more_new_members*⟩ → "," ⟨*new_member*⟩ ⟨*more_new_members*⟩
→ ε
⟨*new_member*⟩ → ⟨*type*⟩ ⟨*new_member_name*⟩
⟨*new_member_name*⟩ → ⟨*id*⟩

⟨***function_statement***⟩ → ⟨*function_name*⟩ ⟨*parameter_list*⟩
⟨*function_name*⟩ → ⟨*id*⟩
⟨*parameter_list*⟩ → "(" ⟨*parameter*⟩ ⟨*more_parameters*⟩ ")"
→ ⟨*parameter*⟩ ⟨*more_parameters*⟩
→ ε
⟨*more_parameters*⟩ → "," ⟨*parameter*⟩ ⟨*more_parameters*⟩
→ ε
⟨*parameter*⟩ → ⟨*id*⟩
→ ⟨*expression*⟩
→ ε

⟨***input_statement***⟩ → "INPUT" ⟨*prompt*⟩ | ⟨*channel*⟩ "," ⟨*var_list*⟩
⟨*prompt*⟩ → ⟨*expression*⟩
⟨*channel*⟩ → "#" ⟨*expression*⟩
⟨*var_list*⟩ → ⟨*var*⟩ ⟨*more_vars*⟩
⟨*more_vars*⟩ → "," ⟨*var*⟩ ⟨*more_vars*⟩
→ ε
⟨*var*⟩ → ⟨*id*⟩

⟨***print_statement***⟩ → "PRINT" [⟨*channel*⟩ ","] ⟨*expression_list*⟩
⟨*expression_list*⟩ → ⟨*expression*⟩ ⟨*more_expressions*⟩
⟨*more_expressions*⟩ → "," ⟨*expression*⟩ ⟨*more_expressions*⟩
→ ε

⟨***open_statement***⟩ → "**OPEN**" ⟨*channel*⟩ "**,**" ⟨*mode*⟩ "**,**" ⟨*filename*⟩
⟨*mode*⟩ → ⟨*expression*⟩
⟨*filename*⟩ → ⟨*expression*⟩

⟨***close_statement***⟩ → "**CLOSE**" [⟨*channel*⟩]

⟨***translate_statement***⟩ → "**TRANSLATE**" "**(**"⟨*expression*⟩"**)**"

⟨***delete_statement***⟩ → "**DELETE**" ⟨*delete_list*⟩
⟨***delete_list***⟩ → ⟨*delete_expression*⟩ ⟨*more_delete_expressions*⟩
⟨*more_delete_expressions*⟩ → "**,**" ⟨*delete_expression*⟩ ⟨*more_delete_expressions*⟩
→ ε
⟨*delete_expression*⟩ → ⟨*expression*⟩

⟨***symbolic_statement***⟩ → "**SYMBOL**" ⟨*expression*⟩

⟨***system_statement***⟩ → "**SYSTEM**" ⟨*expression*⟩

⟨***local_statement***⟩ → "**LOCAL**" ⟨*expression*⟩

⟨***global_statement***⟩ → "**GLOBAL**"

⟨***function_definition***⟩ → "**DEFINE**" ⟨*return_type*⟩ ⟨*new_function_name*⟩ ⟨*param_list*⟩
⟨*statement_list*⟩
"**END_DEFINE**"
⟨*return_type*⟩ → ⟨*type*⟩
⟨*new_function_name*⟩ → ⟨*id*⟩
→ ε
⟨*param_list*⟩ → "**(**"⟨*param*⟩ ⟨*more_params*⟩"**)**"
→ "**(**" "**)**"
→ ε
⟨*more_params*⟩ → "**,**" ⟨*param*⟩ ⟨*more_params*⟩
→ ε
⟨*param*⟩ → ⟨*param_type*⟩ ⟨*param_name*⟩
⟨*param_type*⟩ → ⟨*type*⟩
⟨*param_name*⟩ → ["**&**"] ⟨*id*⟩
→ ["**&**"] ⟨*id*⟩ "**[**" "**]**"

⟨***statement_list***⟩ → ⟨*statement*⟩ ⟨*more_statements*⟩
→ ε
⟨*more_statements*⟩ → ⟨*statement*⟩ ⟨*more_statements*⟩
→ ε

⟨*statement*⟩ → ⟨*declaration_statement*⟩
→ ⟨*external_declaration_statement*⟩
→ ⟨*assignment_statement*⟩
→ ⟨*function_statement*⟩
→ ⟨*conditional_statement*⟩
→ ⟨*iterative_statement*⟩
→ ⟨*return_statement*⟩
→ ⟨*break_statement*⟩
→ ⟨*input_statement*⟩
→ ⟨*print_statement*⟩
→ ⟨*open_statement*⟩
→ ⟨*close_statement*⟩
→ ⟨*translate_statement*⟩
→ ⟨*delete_statement*⟩
→ ⟨*local_statement*⟩
→ ⟨*global_statement*⟩

⟨***external_declaration_statement***⟩ → "**EXTERN**" ⟨*declaration_statement*⟩

⟨***conditional_statement***⟩ → ⟨*if_statement*⟩
→ ⟨*switch_statement*⟩

⟨*if_statement*⟩ → "**IF**" "**(**"⟨*expression*⟩"**)**" ⟨*statement_list*⟩
[{"**ELSEIF**" "**(**"⟨*expression*⟩"**)**" ⟨*statement_list*⟩}]
["**ELSE**" ⟨*statement_list*⟩]
"**ENDIF**"

⟨*switch_statement*⟩ → "**SWITCH**" "**(**"⟨*expression*⟩"**)**"
"**CASE**" "**(**"⟨*expression*⟩"**)**" ⟨*statement_list*⟩
["**CASE**" "**(**"⟨*expression*⟩"**)**" ⟨*statement_list*⟩]
["**DEFAULT**" ⟨*statement_list*⟩]
"**ENDSWITCH**"

⟨***iterative_statement***⟩ → ⟨*for_statement*⟩
→ ⟨*while_statement*⟩
→ ⟨*repeat_statement*⟩

⟨*for_statement*⟩ → "**FOR**" "**(**"⟨*assignment_list*⟩";"⟨*expression*⟩";"⟨*assignment_list*⟩"**)**"
⟨*statement_list*⟩
"**NEXT**"

⟨*while_statement*⟩ → "**WHILE**" "**(**"⟨*expression*⟩"**)**"
⟨*statement_list*⟩
"**ENDWHILE**"

⟨*repeat_statement*⟩ → "**REPEAT**"
⟨*statement_list*⟩
"**UNTIL**" "**(**"⟨*expression*⟩"**)**"

⟨***return_statement***⟩ → "**RETURN**" [⟨*expression*⟩]

⟨***break_statement***⟩ → "**BREAK**"

⟨***expression***⟩ → ⟨*term1*⟩ ⟨*moreterm1s*⟩
⟨*moreterm1s*⟩ → "**OR**" ⟨*term1*⟩ ⟨*moreterm1s*⟩
→ ε
⟨*term1*⟩ → ⟨*term2*⟩ ⟨*moreterm2s*⟩
⟨*moreterm2s*⟩ → "**AND**" ⟨*term2*⟩ ⟨*moreterm2s*⟩
→ ε
⟨*term2*⟩ → ⟨*term3*⟩ ⟨*moreterm3s*⟩
⟨*moreterm3s*⟩ → "=" | "<>" ⟨*term3*⟩ ⟨*moreterm3s*⟩
→ ε
⟨*term3*⟩ → ⟨*term4*⟩ ⟨*moreterm4s*⟩
⟨*moreterm4s*⟩ → "<" | "<=" | ">" | ">=" ⟨*term4*⟩ ⟨*moreterm4s*⟩
→ ε
⟨*term4*⟩ → ⟨*term5*⟩ ⟨*moreterm5s*⟩
⟨*moreterm5s*⟩ → "+" | "-" ⟨*term5*⟩ ⟨*moreterm5s*⟩
→ ε
⟨*term5*⟩ → ⟨*term6*⟩ ⟨*moreterm6s*⟩
⟨*moreterm6s*⟩ → "*" | "/" ⟨*term6*⟩ ⟨*moreterm6s*⟩
→ ε
⟨*term6*⟩ → "+" | "-" | "**NOT**" ⟨*term7*⟩
→ ⟨*term7*⟩
⟨*term7*⟩ → ⟨*term8*⟩ ["^" ⟨*term6*⟩]
⟨*term8*⟩ → "(" ⟨*expression*⟩ ")"
→ ⟨*number*⟩
→ ⟨*sliteral*⟩
→ ⟨*identifier*⟩
→ ⟨*function_statement*⟩
→ "**SIN**" | "**COS**" | "**TAN**" | "**ASIN**" | "**ACOS**" | "**ATAN**" | "**SINH**" | "**COSH**" | "**TANH**" | "**ABS**" | "**SQRT**" | "**LOG**" | "**LN**" | "**EXP**"

⟨***type***⟩ → "**FLOAT**" | "**INTEGER**" | "**STRING**" | "***user defined type***"

⟨***id***⟩ → "***identifier***"

⟨***number***⟩ → "***unsigned integer***" | "***unsigned real***"

⟨***sliteral***⟩ → "***string***"

The following portion of the grammar is embedded in the lexical analyzer and included here for reference only:

⟨*id*⟩	→	⟨*letter*⟩ {⟨*letter*⟩ | ⟨*digit*⟩}
⟨*number*⟩	→	⟨*integer*⟩ ["."] [⟨*exponent*⟩]
	→	[⟨*integer*⟩] "." ⟨*integer*⟩ [⟨*exponent*⟩]
⟨*integer*⟩	→	⟨*digit*⟩ {⟨*digit*⟩}
⟨*exponent*⟩	→	"E" | "e" [⟨*sign*⟩] ⟨*integer*⟩
⟨*sign*⟩	→	"+" | "-"
⟨*digit*⟩	→	"0" | "1" | "2" | ... | "9"
⟨*letter*⟩	→	"A" | "B" | ... | "Z" | | "_" | | "a" | "b" | ... | "z"
⟨*arithmetic*⟩	→	"+" | "-" | "*" | "/" | "^"
⟨*relational*⟩	→	"<" | "<=" | ">" | ">="
⟨*equality*⟩	→	"=" | "<>"
⟨*string_literal*⟩	→	"""{⟨*char*⟩}"""
⟨*char*⟩	→	*any legal character*; ASCII codes 0-127

4. A designation of one of the nonterminals in part 2 as a *starting nonterminal* from which all others can be generated by systematic application of the rules in part 3.

   Starting nonterminal is: ⟨***command_stream***⟩

4. IMPLEMENTATION

Engineering examples, performance evaluation, and application integration are discussed in the following sections. It's recommended that the reader study the Language and Compiler Guide in the Appendix to familiarize themselves with the language and program usage before proceeding with this section.

4.1 Examples Solving Engineering Problems

Several engineering examples are presented in the following sections to illustrate the current capabilities of the translator and system language. The TRANSLATE command is used in some of the examples shown here to exploit the translator's reentrant capability and give the reader some insight into this unusual feature. However, until pointers are included in the system language, efficient use of the TRANSLATE command cannot be attained. Nevertheless, the benefits to the user are such that it was decided to use this feature in the examples that follow.

4.1.1 Four-Bar Crank Mechanism

This example presents a program to solve for the output angles of a four-bar crank mechanism given a set of input angles. Freudenstein's equation is solved numerically using the Newton-Raphson method. For a complete description of the Newton-Raphson method, derivation of Freudenstein's equation, and solution to the four-bar crank mechanism problem, see James, Smith, and Wolford.[6]

Problem: A typical four-bar crank mechanism is shown in Figure 4.1. Calculate the output angles ϕ for a 360° rotation of the input crank in 5° increments starting at $\theta = 0°$. Use the following link lengths: a = 1.0, b = 2.0, c = 2.0, d = 2.0

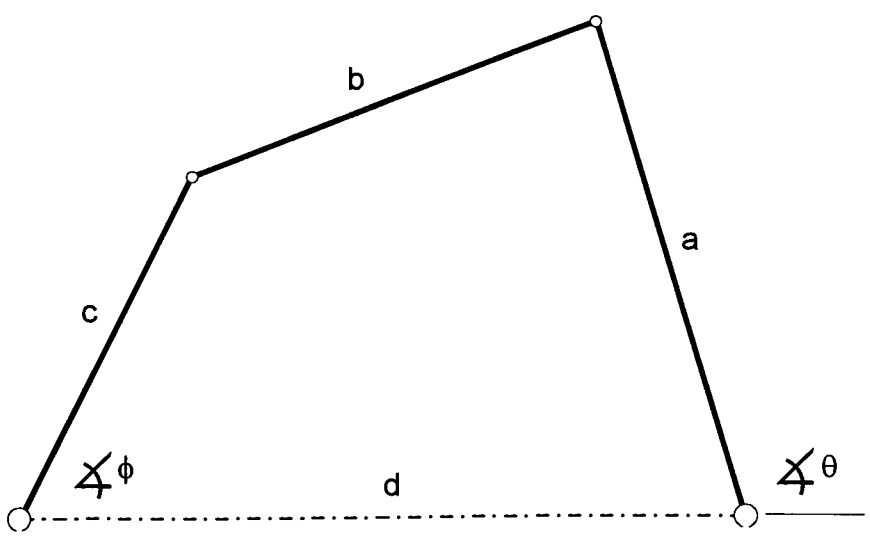


Figure 4.1 Four-bar crank mechanism.

Theory: Calculation of ϕ for a given input angle θ is found by solving Freudenstein's equation:

$$R_1*\cos\theta - R_2*\cos\phi + R_3 - \cos(\theta - \phi) = 0 \qquad \textbf{Equation 4.1}$$

where:

$R_1 = d/c$
$R_2 = d/a$
$R_3 = (d^2 + a^2 - b^2 + c^2)/2*c*a$

In order to solve Equation 4.1, an iterative approach using the Newton-Raphson method is implemented.

Solution: This problem is easily solved using the Newton-Raphson method. Program code to solve the problem is shown in Figure 4.2.

```
// Program to calculate output crank angles in a four-bar mechanism using the
// Newton-Raphson method to solve Freudenstein's equation.
//
// a              length of input crank, in
// b              length of coupler link, in
// c              length of output link, in
// d              length of fixed link, in
// delta_theta    increment of input angle, deg
// theta          value of input angle, deg
// theta_max      maximum value of input angle, deg
// R1,R2,R3       constants calculated from link lengths
// phi            value of output angle, deg and radians
// new_phi        improved value of output angle, deg and radians
// f1             f(phi)=R1*cos(theta)-R2*cos(phi)+R3-cos(theta-phi)
// f0             f'(phi)=R2*sin(phi)-sin(theta-phi)
// epsilon        accuracy check value, radians
```

Figure 4.2 Program to solve four-bar crank mechanism problem.

```
DEFINE bar4()
  FLOAT a,b,c,d                          // variable declarations
  FLOAT delta_theta,theta,theta_max      //
  FLOAT R1,R2,R3,phi,new_phi             //
  FLOAT f1,f0,epsilon                    //
  INTEGER i                              //
  a = 1.0, b = 2.0, c = 2.0, d = 2.0     // variable initializations
  delta_theta = 5.0                      //
  theta = 0.0                            //
  theta_max = 360.0                      //
  phi = 41.0                             //
  epsilon = 0.00001                      //
  R1 = d/c                               // constant calculatons
  R3 = (d^2.0+a^2.0-b^2.0+c^2.0)/(2.0*c*a) //
  theta = theta*0.01745329               // convert angles to radians
  theta_max = theta_max*0.01745329       //
  phi = phi*0.01745329                   //
  delta_theta = delta_theta*0.01745329   //
  FOR(i=1; theta<=theta_max ; i=i+1, theta=theta+delta_theta)
    f1 = R1*COS(theta)-R2*COS(phi)+R3-COS(theta-phi)   // calc Freudenstein eq
    f0 = R2*SIN(phi)-SIN(theta-phi)                    // calc derivative
    new_phi = phi-f1/f0                                // calc improved phi val
    WHILE((ABS(new_phi-phi)-epsilon) > 0)              // iterate calcs until phi converges
      phi = new_phi                                    // with accuracy specified
      f1 = R1*COS(theta)-R2*COS(phi)+R3-COS(theta-phi)
      f0 = R2*SIN(phi)-SIN(theta-phi)
      new_phi = phi-f1/f0
    ENDWHILE
    PRINT theta/0.01745329,",",new_phi/0.01745329
    phi = new_phi                        // approximate next output angle
  NEXT
END_DEFINE
```

Figure 4.2 (cont.) Program to solve four-bar crank mechanism problem.

To run the program, load the file *4bar.mac*. At the command prompt type:

com> bar4() <ENTER>

The program will run and print the results to screen.

Results: Results of calculations are graphed in Figure 4.3.

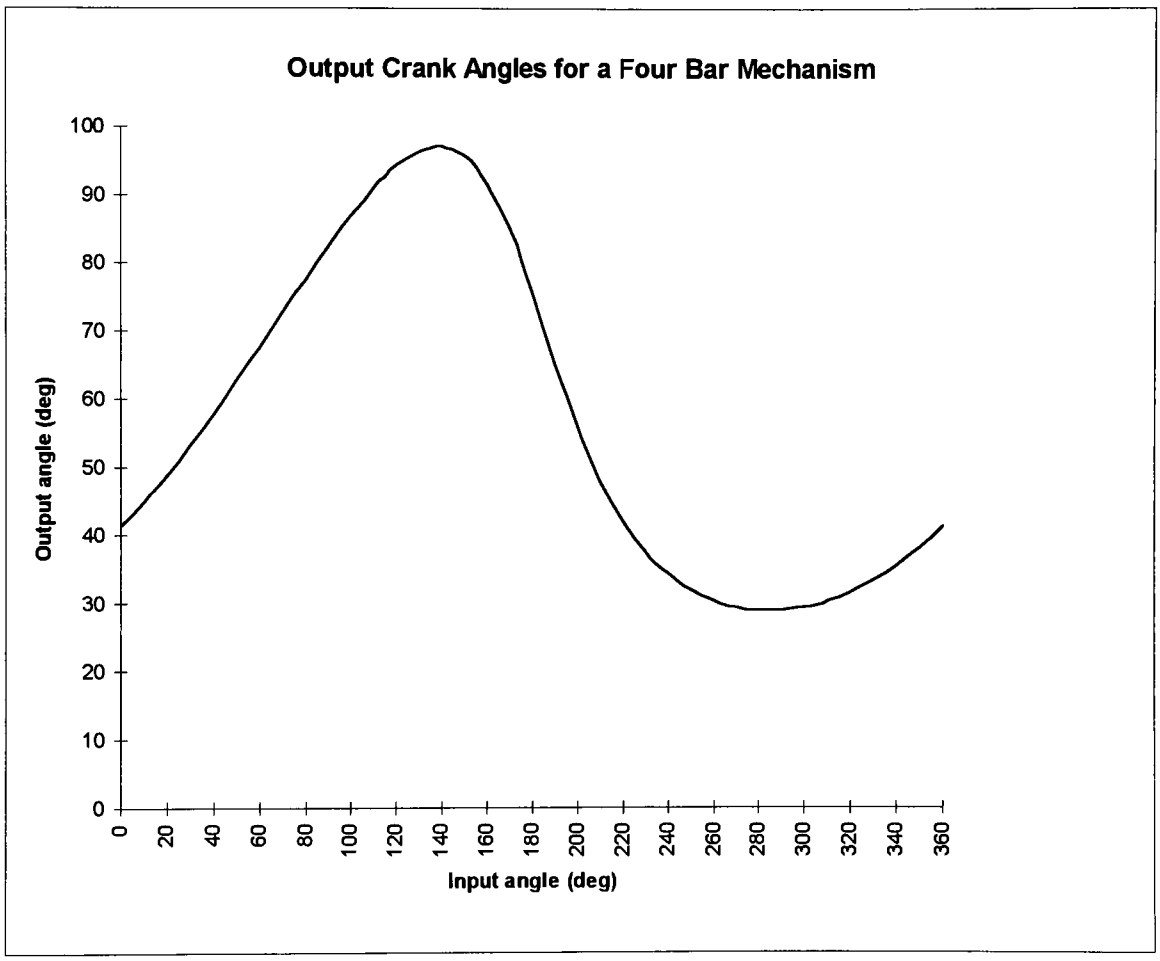


Figure 4.3 Graph of four-bar crank mechanism output angles.

4.1.2 Material Library

This example presents a method for incorporating a materials library into an application environment. A materials library should be storable on disk for retrieval at the users discretion and also provide a mechanism for insertion of new materials. To this end, a single macro file *matlib.mac* is created containing the following:

- a TYPE DEFINITION section for creation of a user-defined type called **material**
- a FUNCTION DEFINITION section for creation of the function **newmat()** which is used to insert new materials into the library
- a MATERIALS section used for declaration and assignment of materials

Figure 4.4 is a listing of the program code to create and maintain a materials library.

```
// TYPE DEFINITION section
TYPEDEF material
{
 FLOAT kx,
 FLOAT ky,
 FLOAT kz,
 FLOAT cp
}
// FUNCTION DEFINITION section
DEFINE newmat()
  STRING name,stream,spc
  FLOAT kx,ky,kz,cp
  INPUT "Enter new material name: ",name
  INPUT "Enter kx: ",kx
  INPUT "Enter ky: ",ky
  INPUT "Enter kz: ",kz
  INPUT "Enter cp: ",cp
  spc = " "
  stream = "material "+name
  stream = stream+spc+name+".kx = "+FTOA(kx)
  stream = stream+spc+name+".ky = "+FTOA(ky)
  stream = stream+spc+name+".kz = "+FTOA(kz)
  stream = stream+spc+name+".cp = "+FTOA(cp)
  TRANSLATE(stream)
  OPEN #1,"a","matlib.mac"
  PRINT #1,""
  PRINT #1,"material "+name
  PRINT #1,name+".kx = ",kx
  PRINT #1,name+".ky = ",ky
  PRINT #1,name+".kz = ",kz
  PRINT #1,name+".cp = ",cp
  CLOSE #1
END_DEFINE
```

Figure 4.4 Material library program.

```
// MATERIALS section
material steel
steel.kx = 9.4
steel.ky = 9.4
steel.kz = 9.4
steel.cp = 0.11

material aluminum
aluminum.kx = 118
aluminum.ky = 118
aluminum.kz = 118
aluminum.cp = 0.214

material copper
copper.kx = 223
copper.ky = 223
copper.kz = 223
copper.cp = 0.0915

material lead
lead.kx = 20
lead.ky = 20
lead.kz = 20
lead.cp = 0.031

material nickel
nickel.kx = 52
nickel.ky = 52
nickel.kz = 52
nickel.cp = 0.1065

material silver
silver.kx = 235
silver.ky = 235
silver.kz = 235
silver.cp = 0.0559
```

Figure 4.4 (cont.) Material library program.

Loading the file *matlib.mac* causes the following to happen:

1. The user-defined type **material** is added to the system.
2. The function **newmat()** is compiled into the system and is available to the user for adding new materials to the library.
3. The variables steel, aluminum, copper, lead, nickel, and silver are created and assigned values.

Once loaded, any of the material variables can be used. For example, to calculate the heat conduction thru a plain wall, Fourier's equation is used;

$Q = -k*A*(T_2-T_1)/\Delta x$

If the variables Q, A, T1, T2, and deltax have been defined, any material can be used in the expression for k as illustrated in the assignment statement below:

```
LET Q =  -steel.kx*A*(T2-T1)/deltax
```

To add a new material to the library, the user invokes the function **newmat()**.

This function interactively prompts the user for 5 inputs:

```
Enter new material name: _
Enter kx: _
Enter ky: _
Enter kz: _
Enter cp: _
```

After entering the new material name and property values, the function appends the new material declaration and assignments to the file matlib.mac using the OPEN #, PRINT #, and CLOSE # commands. This action alone, however, does not load the new material into the system. To accomplish this without having to reload the file matlib.mac, the TRANSLATE command is used. A string variable is created consisting of a declaration statement to declare the new material variable and 4 assignment statements to assign property values to the new material variable. This string is then processed using the TRANSLATE command and the new material is added to the system.

This is a simple example of using the reentrant capability of the translator via the TRANSLATE command.

4.1.3 Integration Solver

This example presents a simple function to perform integration using the Trapezoidal method for approximating the area under a curve. It illustrates a more complex use of the TRANSLATE command to compile functions passed as string variables. It will also point out the need for the inclusion of pointer variables in the system language which, at present, is absent.

Figure 4.5 is a listing of the macro file *integrat.mac*. Loading the file into the system compiles and makes available to the user a single function **INTEGRATE(a,b,n,fx)**. This function will return the result of integrating a function given:

a lower limit of integration
b upper limit of integration
n number of intervals to divide up area of integration for Trapezoidal method
fx string which is the function to be integrated

For example, to perform the integration $\int x^2+3\ dx$ from x= 1 to x =3 and print the result to the screen, type the following at the command line:

```
com> PRINT INTEGRATE(1,3,100,"x^2+3") <ENTER>
14.6668
com> _
```

The result yields 14.6668, a close approximation to the actual value of 14 2/3.

```
DEFINE FLOAT INTEGRATE(FLOAT a, FLOAT b, INTEGER n, STRING fx)
  FLOAT area
  STRING function, trapezoid
  function  = "DEFINE FLOAT _fx(FLOAT x) RETURN "+fx+" END_DEFINE"
  trapezoid = "DEFINE FLOAT _INTEGRATE(FLOAT a, FLOAT b, INTEGER n)
                 FLOAT x,h,sum,area
                 INTEGER i
                 h=(b-a)/n
                 sum=0
                 x=a+h
                 FOR(i=2; i<=n; i=i+1)
                   sum=sum+_fx(x)
                   x=x+h
                 NEXT
                 area=h/2.0*(_fx(a)+2.0*sum+_fx(b))
                 RETURN area
               END_DEFINE"
  TRANSLATE(function)
  TRANSLATE(trapezoid)
  LOCAL "INTEGRATE"
  TRANSLATE("area=_INTEGRATE(a,b,n)")
  GLOBAL
  DELETE "_INTEGRATE","_fx"
  RETURN area
END_DEFINE
```

Figure 4.5 Numerical integration program using the Trapezoidal method.

To help understand the program, a complete step by step analysis is presented below:

1. **DEFINE FLOAT INTEGRATE(FLOAT a, FLOAT b, INTEGER n, STRING fx)**
   Start of function definition *INTEGRATE* which takes as parameters 2 floating point objects, 1 integer, and 1 string and returns a floating point value.

2. **FLOAT area**
   Declaration of local floating point variable *area*.

3. **STRING function, trapezoid**
   Declaration of 2 local string variables *function* and *trapezoid*.

4. **function = "DEFINE FLOAT _fx(FLOAT x) RETURN "+fx+" END_DEFINE"**
   String variable assignment; local string variable *function* is assigned the result of a string expression involving the concatenation of the string literal *"DEFINE FLOAT _fx(FLOAT x) RETURN "*, the string variable *fx*, and the string literal *" END_DEFINE"*.

   It is important to realize that the string assigned to *function* is just that, a string. For the example presented at the beginning of this section, *function* would be assigned the following string:

   function = "DEFINE FLOAT _fx(FLOAT x) RETURN x^2+3 END_DEFINE"

   This string contains a syntactically correct definition of the function _fx to evaluate x^2+3, but no such function exists in the system when INTEGRATE() is compiled.

5. **trapezoid = "DEFINE FLOAT _INTEGRATE(FLOAT a, FLOAT b, INTEGER n)**

```
FLOAT x,h,sum,area
INTEGER i
h=(b-a)/n
sum=0
x=a+h
FOR(i=2; i<=n; i=i+1)
    sum=sum+_fx(x)
    x=x+h
NEXT
area=h/2.0*(_fx(a)+2.0*sum+_fx(b))
RETURN area
END_DEFINE"
```

   String variable assignment; local string variable *trapezoid* is assigned a string literal. Again, as in step 4, it is important to realize that the string assigned to *trapezoid* is just that, a string. The string contains a syntactically correct definition of the function _INTEGRATE, but no such function exists in the system when INTEGRATE() is compiled. _INTEGRATE is a function to perform integration using the trapezoidal method and integrand _fx.

6. **TRANSLATE(function)**
   Call to system command TRANSLATE(). At runtime, a call to TRANSLATE(function) will suspend the current operation of the back end interpreter, save the register contents, preserve the stack contents, and then reenter the translator to process the contents of the string variable *function*. Once this operation is complete, control returns to the back end interpreter and execution of INTEGRATE() resumes. At this point, a new function called _fx() has been compiled into the system.

7. **TRANSLATE(trapezoid)**
   Call to system command TRANSLATE(). At runtime, a call to TRANSLATE(trapezoid) will suspend the current operation of the back end interpreter, save the register contents, preserve the stack contents, and then reenter the translator to process the contents of the string variable *trapezoid*. Once this operation is complete, control returns to the back end interpreter and execution of INTEGRATE() resumes. At this point, a new function called _INTEGRATE() has been compiled into the system. Note that _INTEGRATE() contains calls to the function _fx(), where _fx() evaluates the function that was passed as a string to INTEGRATE().

8. **LOCAL "INTEGRATE"**
   Call to system command LOCAL "*function name*". This command sets the symbol table scope local to the function INTEGRATE().

9. **TRANSLATE("area=_INTEGRATE(a,b,n)")**
   Call to system command TRANSLATE(). At runtime, this call to TRANSLATE() will suspend the current operation of the back end interpreter, save the register contents, preserve the stack contents, and then reenter the translator to process the string literal "area = _INTEGRATE(a,b,n)". Translation of this string results in a call to the function _INTEGRATE(a,b,n) and assignment of the result to *area*. Note that a, b, n, and area are all local variables to the function INTEGRATE(). This is the reason that in step 8, the symbol table scope was set to LOCAL. Normally, when entering the translator, symbol table scope is global. This means that objects inside of functions are hidden, and as a result, area, a, b, and n would not be found. The LOCAL command circumvents this problem.

   Once this operation is complete, control returns to the back end interpreter and execution of INTEGRATE() resumes. At this point, *area* contains the result of the integration.

10. **GLOBAL**
    Call to system command GLOBAL. This command sets the symbol table scope back to global, the default mode.

11. **DELETE "_INTEGRATE","_fx"**
    Call to system command DELETE "*object name*". This command deletes the functions _INTEGRATE and _fx from the system.

12. **RETURN** area
    Call to system command RETURN *expression*. Terminates execution of the function INTEGRATE() and returns the value of *area*.

13. **END_DEFINE**
    End of function definition INTEGRATE().

At present, *pointers* are not implemented in the language. This is the reason why most of the code to do the integration has to be compiled each time INTEGRATE() is called. To understand this, you must have a clear distinction between *compile time* and *run time* actions. At compile time, a statement which includes a call to a function results in a search of the symbol table for that function to retrieve its address. Since _fx() does not exist at compile time for the function INTEGRATE(), a statement that includes a call to _fx() in INTEGRATE() will result in an error. For this reason, the code steps to perform the integration cannot include a call to _fx() when INTEGRATE() is compiled.

At runtime, the back end interpreter executes INTEGRATE() by stepping thru its threaded code list. When the code to execute the TRANSLATE() command is executed, execution of INTEGRATE() is temporarily suspended, and the translator is reentered to process the string passed to the TRANSLATE() command. _fx() is compiled into the system and control returns to resume execution of INTEGRATE(). At this point, _fx() is in the system and has a symbol table entry, but INTEGRATE() has no knowledge of it because it didn't exist at compile time for INTEGRATE(). The next call to TRANSLATE() processes the string defining the function _INTEGRATE(). This function contains calls to _fx() which are valid because _fx() is a function in the system at compile time for _INTEGRATE().

Finally, to execute _INTEGRATE() to perform the actual integration, the TRANSLATE() command must once more be used. Again, this is because _INTEGRATE() did not exist at compile time for INTEGRATE().

When pointers are finally incorporated into the language, it will be much easier and more efficient to program using the TRANSLATE() command. Figure 4.6 shows a possible scenario for programming the integration routine using pointers. This scenario extends the current language to include the following:

- Function pointers; [***type***] (****function_name***)([***parameter_type_list***])
- Runtime routine to get the address of an object; **ADR("*object_name*")**.

In this scenario, function pointer **f** is declared as **FLOAT (*f)(FLOAT)**; a pointer to a function that takes one floating point parameter and returns a floating point value. Once declared, **f** can be used anywhere in INTEGRATE() because at compile time, **f** exists. At runtime, a call to **f** will result in the execution of whatever function it points to at that time.

During execution of INTEGRATE(), the call to TRANSLATE() will compile _fx() into the system. After that, a call to the runtime routine **ADR()** will get the address of _fx() and assign it to **f**. Since _fx() now exists, ADR() will find it's address when it searches the symbol table.

Note in Figure 4.6 that calls to LOCAL and GLOBAL have been eliminated. These commands were specifically put into the language to compensate for the lack of pointers, and should be removed from the language once pointers become available.

```
DEFINE FLOAT INTEGRATE(FLOAT a, FLOAT b, INTEGER n, STRING fx)
  FLOAT x,h,sum,area,(*f)(FLOAT)
  INTEGER i
  TRANSLATE("DEFINE FLOAT _fx(FLOAT x) RETURN "+fx+" END_DEFINE")
  f=ADR("_fx")
  h=(b-a)/n
  sum=0
  x=a+h
  FOR(i=2; i<=n; i=i+1)
    sum=sum+f(x)
    x=x+h
  NEXT
  area=h/2.0*(f(a)+2.0*sum+f(b))
  DELETE "_fx"
  RETURN area
END_DEFINE
```

Figure 4.6 Hypothetical integration routine using pointers.

4.1.4 Fourth-Order Runge-Kutta DEQ Solver

This example presents a program to solve a set of first-order differential equations using a fourth-order Runge-Kutta technique. An input data file is utilized to specify run parameters and an output file is used to store the results. The TRANSLATE command is again used to allow specification of differential equations in the input data file. This eliminates the user's burden of having to write and compile functions each time a new set of differential equations is analyzed as is typical with FORTRAN or BASIC. The user only modifies the data file, not the program code.

The theory behind solutions to differential equations and the algorithm used here will not be discussed, however, the reader is referred to Burden and Faires.[7]

Problem: The flight of a rocket is governed by the second-order, nonlinear differential equation shown below:

$$d^2y/dt^2 + g*K*(dy/dt)^2/W+g-g*T/W = 0$$

where: y = rocket displacement
g = gravitational constant
T = rocket thrust force
W = weight of rocket and fuel
K = drag coefficient

Objective: Solve for the displacement, velocity, and acceleration of the rocket over the time interval t=0 to t=60s given the following data:

g = 32.17 ft/s^2
T = 7000 lbs
W = W(t) = 3000-40t lbs
K = 0.008 $lb\text{-}s^2/ft^2$

Initial conditions for the problem are: $y(t)|_{t=0} = 0$, $dy(t)/dt|_{t=0} = 0$

Solution: This problem is easily solved using the classical 4th-order Runge-Kutta numerical method. Figure 4.7 lists the input data file for this problem. The file format is typical for any problem, and consists of the following sections:

DESCRIPTION	Comment section to write a description of the problem. This section is optional, but highly desirable.
VARIABLE	Section to specify dependent and independent variables used in the problem. **Syntax:** *"dependent_variable","independent_variable"*
INTERVAL	Section to specify initial and final independent variable values and number of steps over the interval. **Syntax:** *intitial_value, final_value, steps*

FUNCTION — Section to define any functions required if differential equations have variable coefficients.

Syntax: *number_of_functions* // specify 0 if there are no functions.
"function_definition" // one function per line
:

CONSTANT — Section to define any constants used in the differential equations.

Syntax: *number_of_constants* // specify 0 if there are no constants.
"constant_declaration constant_assignment"
:

DEQ — Section to specify initial values and define differential equations

Syntax: *number_of_diffeqs*
intitial_value,"differential_equation"
:

```
// ************************** DESCRIPTION SECTION ************************//
// Problem:      Solution of a rocket's flight.
// Equations:    y"+gKy'^2/W+g-gT/W = 0
//               where:  y   rocket displacement, ft
//                       g   gravitational constant, 32.17 ft/s2
//                       T   rocket thrust force, 7000 lbs
//                       W   weight of rocket and fuel, W(t) = 3000-40t lbs
//                       K   drag coefficient, 0.008 lb-s2/ft2
// Intial cond:  y(t=0) = 0, dy/dt(t=0) = 0
//
// Reduction to system of first-order differential equations :
//               y'  = dy/dt  = y'
//               y" = dy'/dt = -gKy'^2/W-g+gT/W

// ************************** VARIABLE SECTION ***************************//
"y","t"

// ************************** INTERVAL SECTION ***************************//
0.0,60.0,600

// ************************** FUNCTION SECTION ***************************//
1
"DEFINE FLOAT W(FLOAT t) RETURN 3000-40*t END_DEFINE"

// ************************** CONSTANT SECTION ***************************//
3
"FLOAT T T=7000"
"FLOAT g g=32.17"
"FLOAT K K=0.008"

// ************************** DEQ SECTION ********************************//
2
0.0,"y[1]"                                  // y (0) , y'
0.0, "-g*K*y[1]^2/W(t)-g+g*T/W(t)"          // y'(0) , y"
```

Figure 4.7 Input data file for Runge-Kutta program.

Figure 4.9 is a program listing for the example. Figure 4.10 is a rewrite of the program as it would appear when pointers are included into the language and is shown for reference only.

To run the program, load the file *rkutta.mac*. Given input file *rkutta.in* and desired output file *rkutta.out*, at the command prompt type:

com> rkutta("rkutta.in","rkutta.out") <ENTER>

The program will run and print the results to the output file.

Results: The results of the program run are plotted in Figure 4.8. See James, Smith, and Wolford for a solution to this problem using Milne's method.[8]

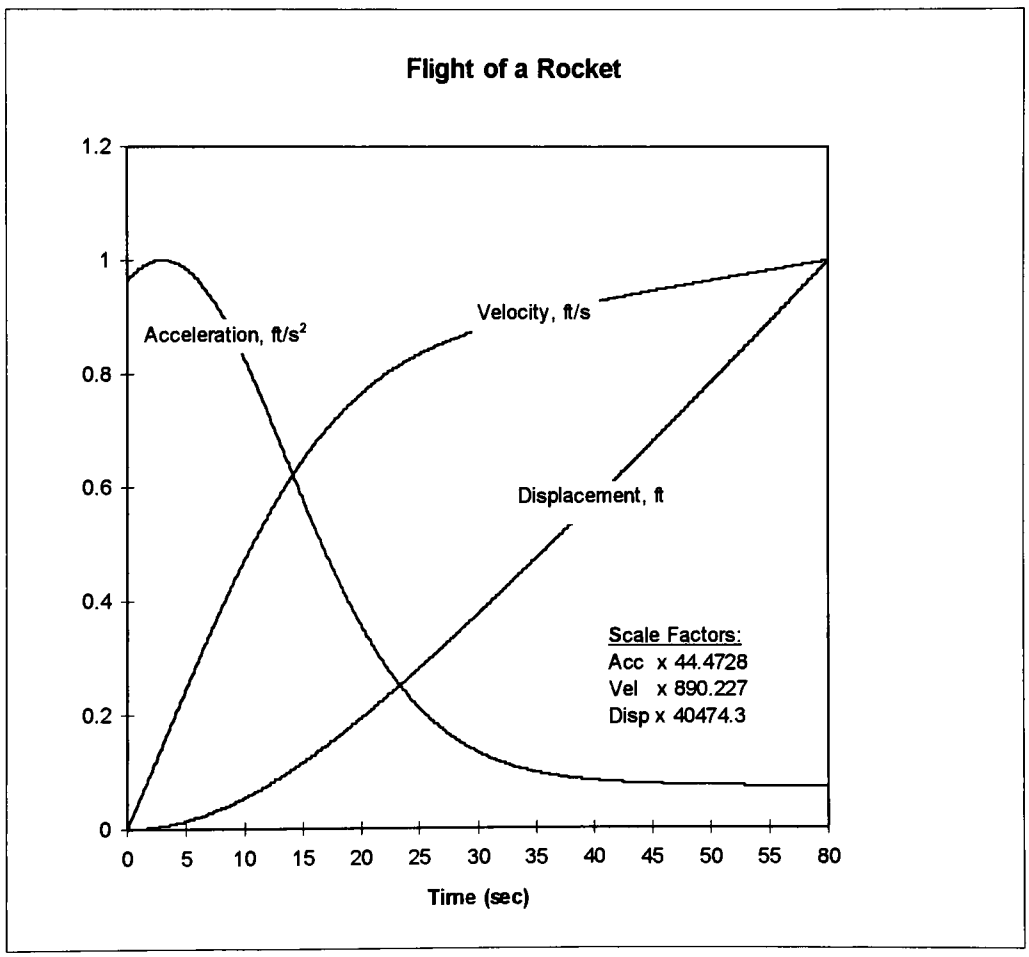


Figure 4.8 Rocket flight characteristics.

```
// Program to solve an nth-order differential equation using the classic 4th-order Runge-Kutta
// method to solve a system of first-order differential equations.

DEFINE rkutta(STRING filein, STRING fileout)
  FLOAT a,b,y[0:9]
  INTEGER m,n,i,nofuncs,nostmts
  STRING dif[0:9],s,dep,indep,func,stmt

/*** Open input file and read in problem parameters. Also compile any supporting functions that
     are required if variable coefficients are present in the differential equations ***/
  OPEN #1,"r",filein              // open input data file
  INPUT #1,dep,indep              // read in dependent and independent variables
  INPUT #1,a,b,n                  // read interval start and ending values, number of interval steps
  INPUT #1,nofuncs                // read in number of variable coefficient functions required
  FOR(i=1;i<=nofuncs;i=i+1)       // read in variable coefficient functions and compile into system
    INPUT #1,func                 //   read function
    TRANSLATE(func)               //   compile function
  NEXT

/*** Build string to define routine to evaluate the first-order differential equation set ***/
  s="DEFINE FLOAT G(INTEGER _j,FLOAT "+dep+"[],FLOAT "+indep+") "
  INPUT #1,nostmts                // read in number of supporting constants for problem
  FOR(i=1;i<=nostmts;i=i+1)       // read in constant declarations and assignments
    INPUT #1,stmt
   s=s+" "+stmt+" "
  NEXT
  INPUT #1,m                  // read in number of first-order differential equations
  FOR(i=0;i<m;i=i+1)          // read in initial values and differential equations
    INPUT #1,y[i],dif[i]
  NEXT
  CLOSE #1
  s=s+"SWITCH(_j) "           // build switch block to evaluate differential equations
  FOR(i=0;i<m;i=i+1)
    s=s+"CASE("+FTOA(i)+")"+" RETURN "+dif[i]+" "
  NEXT
  s=s+"ENDSWITCH END_DEFINE"

/*** Translate string to compile the routine to evaluate the first-order differential equation set ***/
  TRANSLATE(s)
```

Figure 4.9 Program to implement 4th-order Runge-Kutta DEQ solver.

```
/*** String defining the Runge-Kutta routine ***/
  s="DEFINE solver(STRING fileout, FLOAT a,FLOAT b,INTEGER n,INTEGER m,FLOAT u[])
       INTEGER i,j,L,p
       FLOAT x,h,t,k[0:9,4],z[0:9]
       OPEN #1,""w"",fileout
       h=(b-a)/n
       x=a
       PRINT #1,x,"","",u[0],"","",u[1],"","",G(m-1,u[],x)
       FOR(i=1; i<=n; i=i+1)
         FOR(L=1; L<=4; L=L+1)
           FOR(j=0; j<m; j=j+1)
             SWITCH(L)
               CASE(1)
                 t=x
                 FOR(p=0; p<m; p=p+1)
                   z[p]=u[p]
                 NEXT
                 BREAK
               CASE(2)
                 t=x+h/2.0
                 FOR(p=0; p<m; p=p+1)
                   z[p]=u[p]+k[p,1]/2.0
                 NEXT
                 BREAK
               CASE(3)
                 t=x+h/2.0
                 FOR(p=0; p<m; p=p+1)
                   z[p]=u[p]+k[p,2]/2.0
                 NEXT
                 BREAK
               CASE(4)
                 t=x+h
                 FOR(p=0; p<m; p=p+1)
                   z[p]=u[p]+k[p,3]
                 NEXT
                 BREAK
             ENDSWITCH
             k[j,L]=h*G(j,z[],x)
           NEXT
         NEXT
         FOR(j=0; j<m; j=j+1)
           u[j]=u[j]+(k[j,1]+2.0*k[j,2]+2.0*k[j,3]+k[j,4])/6.0
         NEXT
         PRINT #1,x+h,"","",u[0],"","",u[1],"","",G(m-1,z[],x+h)
         x=a+i*h
       NEXT
       CLOSE #1
     END_DEFINE"
```

Figure 4.9 (cont.) Program to implement 4th-order Runge-Kutta DEQ solver.

```
/*** Translate string to compile the Runge-Kutta routine ***/
  TRANSLATE(s)

/*** Solve the set of differential equations ***/
  LOCAL "rkutta"
  TRANSLATE("solver(fileout,a,b,n,m,y[])")
  GLOBAL
  DELETE "G","solver"
END_DEFINE
```

Figure 4.9 (cont.) Program to implement 4th-order Runge-Kutta DEQ solver.

```
DEFINE solver(STRING fileout, FLOAT a,FLOAT b,INTEGER n,INTEGER m,FLOAT u[])
  INTEGER i,j,L,p
  FLOAT x,h,t,k[0:9,4],z[0:9],(*g)(INTEGER,FLOAT,FLOAT)
  g=ADR("G")
  OPEN #1,""w"",fileout
  h=(b-a)/n
  x=a
  PRINT #1,x,"","",u[0],"","",u[1],"","",g(m-1,u[],x)
  FOR(i=1; i<=n; i=i+1)
    FOR(L=1; L<=4; L=L+1)
      FOR(j=0; j<m; j=j+1)
        SWITCH(L)
          CASE(1)
            t=x
            FOR(p=0; p<m; p=p+1)
              z[p]=u[p]
            NEXT
            BREAK
          CASE(2)
            t=x+h/2.0
            FOR(p=0; p<m; p=p+1)
              z[p]=u[p]+k[p,1]/2.0
            NEXT
            BREAK
          CASE(3)
            t=x+h/2.0
            FOR(p=0; p<m; p=p+1)
              z[p]=u[p]+k[p,2]/2.0
            NEXT
            BREAK
          CASE(4)
            t=x+h
            FOR(p=0; p<m; p=p+1)
              z[p]=u[p]+k[p,3]
            NEXT
            BREAK
        ENDSWITCH
        k[j,L]=h*g(j,z[],x)
      NEXT
    NEXT
    FOR(j=0; j<m; j=j+1)
      u[j]=u[j]+(k[j,1]+2.0*k[j,2]+2.0*k[j,3]+k[j,4])/6.0
    NEXT
    PRINT #1,x+h,"","",u[0],"","",u[1],"","",g(m-1,z[],x+h)
    x=a+i*h
  NEXT
  CLOSE #1
END_DEFINE
```

Figure 4.10 Hypothetical Runge-Kutta solver using pointers.

```
DEFINE rkutta(STRING filein, STRING fileout)
  FLOAT a,b,y[0:9]
  INTEGER m,n,i,nofuncs,nostmts
  STRING dif[0:9],s,dep,indep,func,stmt

/*** Open input file and read in problem parameters. Also compile any supporting functions that
     are required if variable coefficients are present in the differential equations ***/

  OPEN #1,"r",filein              // open input data file
  INPUT #1,dep,indep              // read in dependent and independent variables
  INPUT #1,a,b,n                  // read interval start and ending values, number of interval steps
  INPUT #1,nofuncs                // read in number of variable coefficient functions required
  FOR(i=1;i<=nofuncs;i=i+1)       // read in variable coefficient functions and compile into system
    INPUT #1,func                 //   read function
    TRANSLATE(func)               //   compile function
  NEXT

/*** Build string to define routine to evaluate the first-order differential equation set ***/
  s="DEFINE FLOAT G(INTEGER _j,FLOAT "+dep+"[],FLOAT "+indep+") "
  INPUT #1,nostmts                // read in number of supporting constants for problem
  FOR(i=1;i<=nostmts;i=i+1)       // read in constant declarations and assignments
    INPUT #1,stmt
   s=s+" "+stmt+" "
  NEXT
  INPUT #1,m                  // read in number of first-order differential equations
  FOR(i=0;i<m;i=i+1)          // read in initial values and differential equations
    INPUT #1,y[i],dif[i]
  NEXT
  CLOSE #1
  s=s+"SWITCH(_j) "           // build switch block to evaluate differential equations
  FOR(i=0;i<m;i=i+1)
    s=s+"CASE("+FTOA(i)+")"+" RETURN "+dif[i]+" "
  NEXT
  s=s+"ENDSWITCH END_DEFINE"

/*** Translate string to compile the routine to evaluate the first-order differential equation set ***/
  TRANSLATE(s)

/*** Solve the set of differential equations ***/
  solver(fileout,a,b,n,m,y[])

/*** Delete the function created to evaluate the differential equations ***/
  DELETE "G"

END_DEFINE
```

Figure 4.10 (cont.) Hypothetical Runge-Kutta solver using pointers.

4.1.5 Convolution Integral

This example presents a general program to solve for the response of a linear system subjected to an arbitrary forcing function using the convolution integral. As with the previous example, an input data file is utilized to specify run parameters and an output file is used to store the results. The TRANSLATE command is used to allow specification of equations in the input data file; the user only modifies the data file, not the program code.

Problem: Solve for the response of a 2nd-order system with an equation of motion given by:

$d^2y/dt^2 + 2\zeta\omega_0 dy/dt + \omega_0^2 y = m^{-1} f(t)$ where: $\omega_0 = [k/m]^{1/2}$, $\zeta = c*\omega_0/(2*k)$

The unit impulse response for the system is:

$h(t) = [m\omega_0(1-\zeta^2)^{1/2}]^{-1} * e^{-\varpi 0\zeta t} * \sin[(1-\zeta^2)^{1/2}\omega_0 t]$ for $\zeta < 1$

Given a series of pulses with period *T* and duration *dur*, the forcing function $f(t)$ is piecewise defined as follows:

$$f(t) = 0.25(t/20)^2 * \sum_{n=0}^{\infty} [u(t-n*T)*u(n*T+dur-t)] \quad ; 0 \le t < 20$$

$$f(t) = 0.25 * \sum_{n=0}^{\infty} [u(t-n*T)*u(n*T+dur-t)] \quad ; 20 \le t < 40$$

$$f(t) = 0 \quad ; 40 \le t \le 60$$

Parameters:

m = 1.0 slug
ω_0 = 1.0 rad/s
ζ = 0.2
T = 1.0 s
dur = 0.5 s
t_0 = 0.0 s
t_f = 60.0 s

Theory: The response of a linear system to any arbitrary input can be found by considering the input to be a series of impulses and superimposing the individual responses for each impulse. This procedure is called convolution and is represented by the integral shown in Equation 4.2.

$$x(t) = \int_{-\infty}^{t} F(\tau)*h(t-\tau)d\tau \qquad \textbf{Equation 4.2}$$

Solution: This problem is easily solved using the convolution integral. Figure 4.11 lists the input data file for this problem. The input file format is typical for any problem, and consists of the following sections:

TIME STEP	Integration step size to use for Trapezoidal integration routine. **Syntax:** *time_step*
SOLUTION STEP	Solution step size for x(t); must be a multiple of integration step size. **Syntax:** *solution_step*
CONSTANTS	Section to define any floating point constants used in the equations; C is required. **Syntax:** *"constant #1, constant #2, constant #3, ..."*
ASSIGNMENTS	Section to assign values to constants; C is required even if C = 1. **Syntax:** *"constant #1=value, constant #2=value, constant #3=value, ..."*
UNIT IMPULSE RESPONSE	Section to define system response to unit impulse, h(t-s) **Syntax:** *"h(t-s)"*
FORCING FUNCTION	Section to define piecewise forcing function F(s). **Syntax:** *number_of_functions* *interval_start,interval_end,"F(s)"* :

```
// Convolution Integral to solve for the response x(t) to a linear system excited by an arbitrary
// forcing function F(t).
//                              x(t) = C*INTEGRAL(0,t,F(s)*h(t-s)ds)

// *** TIME STEP *** //
0.1   // integration time step for Trapezoidal method

// *** SOLUTION STEP *** //
0.2   // solution step size for x(t) (multiple of integration time step)

// *** CONSTANTS *** //
"Fo,m,wn,zeta,C"  // Constant declarations; C is required

// *** ASSIGNMENTS *** //
"Fo=1,m=1,wn=1,zeta=0.20,C=Fo/(m*wn*SQRT(1-zeta^2))" // Assignments;  C is required

// *** UNIT IMPULSE RESPONSE *** //
"EXP(-zeta*wn*(t-s))*SIN(SQRT(1.0-zeta^2.0)*wn*(t-s))"   // h(t-s)

// *** FORCING FUNCTION *** //
3
0.0,20.0,"(s/20)^2*PULSE(0.25,0.5,1.0,s)"
20.0,40.0,"PULSE(0.25,0.5,1.0,s)"
40.0,60.0,"0.0"
```

Figure 4.11 Input data file for Convolution Integral program.

Figure 4.13 is a program listing for the example.

To run the program, load the file *convolut.mac*. Given input file *convolut.in* and desired output file *convolut.out*, at the command prompt type:

com> CON("convolut.in","convolut.out") <ENTER>

The program will run and print the results to the output file.

Results: The solution for the problem is graphed in Figure 4.12.

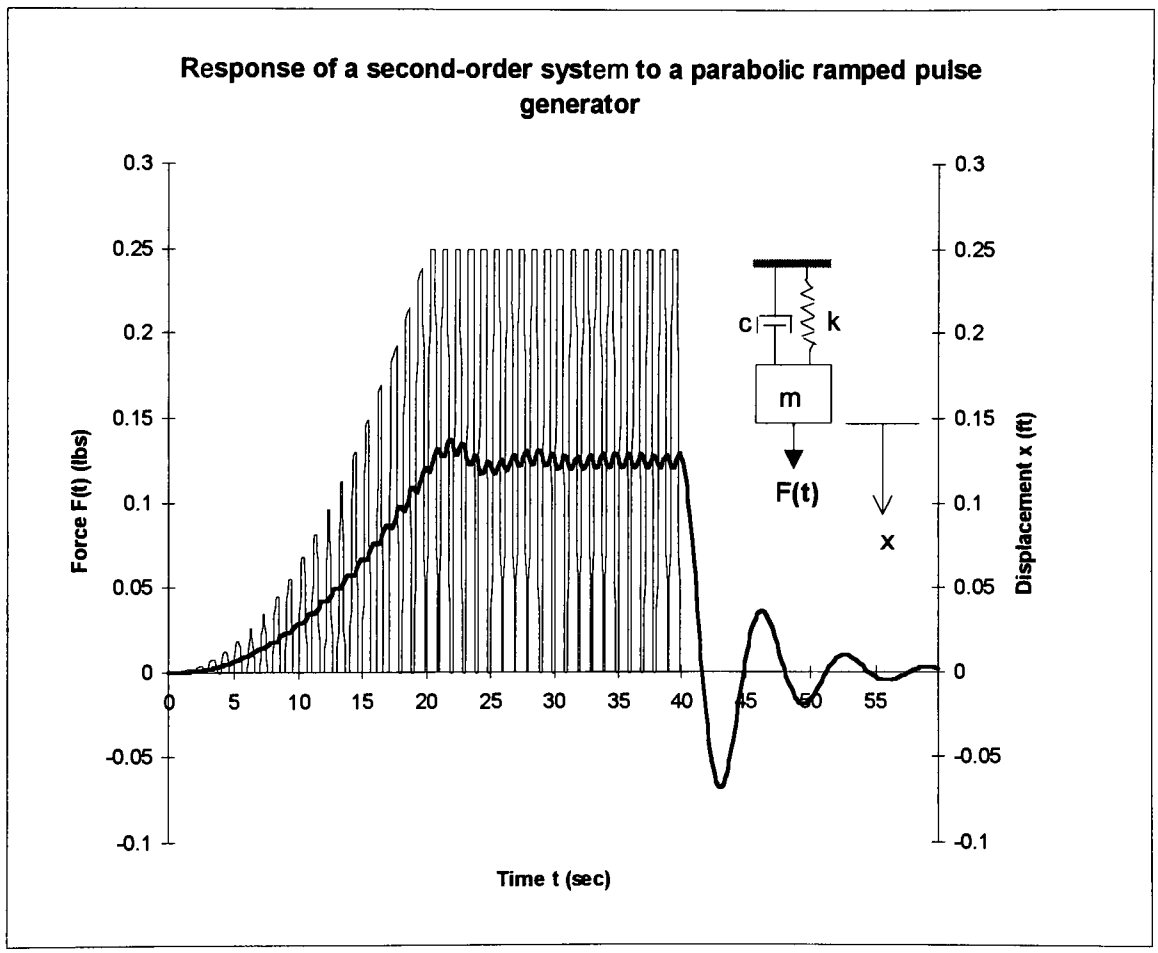


Figure 4.12 Response of 2nd-order system using the Convolution Integral.

```
// User defined type to store piecewise components of F(s); interval start, end, forcing function
TYPEDEF FS {FLOAT t1,FLOAT t2, STRING func}

// *** Pulse generator function *** //
DEFINE FLOAT PULSE(FLOAT mag,FLOAT dur,FLOAT freq,FLOAT s)
  FLOAT threshold,w
  w=pi*freq                    // make frequency of cosine function 1/2 frequency of pulse
  threshold=COS(w*dur/2.0)     // calculate threshold value to turn pulse on
  IF(ABS(COS(w*s-dur/2.0))>=threshold)
    RETURN rnag     // pulse on
  ENDIF
  RETURN 0          // pulse off
END_DEFINE

// *** Convolution Integral function *** //
DEFINE FLOAT CON(STRING filein,STRING fileout)
  FLOAT a,b,step,xstep,area
  INTEGER fsnum,i
  STRING s,dec,asn,hts
  FS fs[10]              // array to store piecewise functions and intervals
  OPEN #1,"r",filein     // open the input data file
  OPEN #2,"w",fileout    // open the output data file
  INPUT #1,step          // read in integration time step
  INPUT #1,xstep         // read in solution step size for x(t)
  INPUT #1,dec           // read in constant declaration list
  dec = "FLOAT "+dec     // prefix constant declaration list with FLOAT type specifier
  INPUT #1,asn           // read in constant assignment list
  INPUT #1,hts           // read in unit input response function h(t-s)
  INPUT #1,fsnum         // read in number of piecewise forcing functions
  FOR(i=1;i<=fsnum;i=i+1)  // read in piecewise forcing functions defining F(s)
    INPUT #1,fs[i].t1,fs[i].t2,fs[i].func  // interval start, interval end, F(s) over interval
  NEXT
  CLOSE #1               // close the input data file
  a=fs[1].t1             // start of solution interval
  b=fs[fsnum].t2         // end of solution interval
  TRANSLATE(dec)     // translate constant declarations into the system
  TRANSLATE(asn)     // translate constant assignments into the system

// Build string defining function Fs() to evaluate piecewise functions for F(s)
  s="DEFINE FLOAT Fs(FLOAT s) "+"EXTERN "+dec+" "
  s=s+"IF(s>="+FTOA(fs[1].t1)+" AND s<="+FTOA(fs[1].t2)+") RETURN "+fs[1].func+" "
  FOR(i=2;i<=fsnum;i=i+1)
    s=s+"ELSEIF(s>="+FTOA(fs[i].t1)+" AND s<="+FTOA(fs[i].t2)+") RETURN "+fs[i].func+" "
  NEXT
  s=s+"ELSE RETURN 0.0"
  s=s+" ENDIF END_DEFINE"
```

Figure 4.13 Convolution Integral program.

```
// Compile Fs() into the system
  TRANSLATE(s)

// Build string defining function Hts() to evaluate h(t-s); compile Hts() into the system
  TRANSLATE("DEFINE FLOAT Hts(FLOAT s,FLOAT t) "+"EXTERN "+dec+" RETURN "+hts+"
                END_DEFINE")

// Build string defining function _INTEGRATE() to perform integration using Trapezoidal
// method; compile _INTEGRATE() into the system
  TRANSLATE("DEFINE FLOAT _INTEGRATE(FLOAT a, FLOAT b, FLOAT step,FLOAT xstep)
                  EXTERN "+dec+" FLOAT s,sum,area,count
                  count=xstep
                  PRINT #2,a,"","",Fs(a),"","",0
                  WHILE(xstep<=b)
                    sum=0.0
                    s=a+step
                    PRINT b,""  "",xstep
                    WHILE(s<=xstep)
                      sum=sum+Fs(s)*Hts(s,xstep)
                      s=s+step
                    ENDWHILE
                    area=C*step/2.0*(Fs(a)*Hts(a,xstep)+2.0*sum+Fs(xstep)*Hts(xstep,xstep))
                    PRINT #2,xstep,"","",Fs(s-step),"","",area
                    xstep=xstep+count
                  ENDWHILE
                  RETURN area
                END_DEFINE")

// Translate expression to do integration
  LOCAL "CON"
  TRANSLATE("area=_INTEGRATE(a,b,step,xstep)")
  GLOBAL
  DELETE "_INTEGRATE","Fs","Hts"    // delete functions
  CLOSE #2          // close the output data file
  RETURN area       // return result of integration
END_DEFINE

// To solve for response x(t), execute function CON()
// CON("input_filename","output_filename")
```

Figure 4.13 (cont.) Convolution Integral program.

4.2 Performance Evaluation

Translator performance is effected by several factors. Excluding any effects attributable to the user's choice of source code constructs or algorithms, the following factors effect compilation and execution:

Compilation speed is dependent on:

- Design and efficiency of the front end translator algorithms and data structures.
- Efficient symbol table routines, in particular, search algorithms and memory allocation routines.
- Design and efficiency of the back end threaded interpreter algorithms and data structures, in particular, the TIL dictionary and supporting search and insertion routines, and stack and register usage.
- Size of symbol table and TIL dictionary at the start of compilation

Execution speed is dependent on:

- Design and efficiency of the threaded interpreter algorithms, especially the inner interpreter routines and stack and register usage.
- Efficiency of primitive source code algorithms.
- Number of primitives.

The single-pass compiler used in the translator is relatively fast when compiling simple macro files. Approximate compilation times for the examples presented earlier are tabulated in Table 4.1. The files were compiled on a 486/33 MHz PC.

Macro File Name	Compilation Time (sec)
4bar.mac	0.11
matlib.mac	0.17
integrat.mac	0.05
rkutta.mac	0.11
convolut.mac	0.16

Table 4.1 Compilation times for a few selected macro files.

The more significant improvements that would contribute to increases in compilation speed are:

- Change the parser design. The current recursive-descent parser relies heavily on recursive function calls which are time consuming processes.
- Use hashing in the symbol table; currently using a linked list with linear searching.
- Purchase a third-party memory management library for the C/C++ compiler. There are more efficient memory allocation routines than are supplied with a standard C/C++ compiler.
- Improve the design, algorithms, and data structures currently implemented in the translator.
- Code critical routines in assembler.

Execution speed is critical for any type of algorithmic programming. A typical TIL written in assembler produces executable code relatively slow compared to optimal assembled code; approximately 100% inefficient.[9] A TIL written in a high-level language increases this inefficiency significantly. The TIL used in this design was written in C/C++ to make the job of development easier and allow portability to different platforms. For these reasons, execution speed is relatively slow compared to a TIL written in assembler.

To gain some insight into actual execution speed, a simple test was done running similar code on this translator, MS-DOS Qbasic, and C/C++. The code executes a while loop 10000 times, performing floating point division, multiplication, and addition during each iteration. Table 4.2 lists the source code for the test.

C/C++ Source Code	Translator Source Code	Qbasic Source Code

void main()
{
  int i;
  double a,b,c;
  i=0;
  a=1.5,b=2.5;
  while(i<10000)
  {
  i=i+1;
  c=a/b+a*b;
  }
}
	DEFINE test()
	  INTEGER i
	  FLOAT a,b,c
	  i=0
	  a=1.5,b=2.5
	  WHILE(i<10000)
	  i=i+1
	  c=a/b+a*b
	  ENDWHILE
	END_DEFINE
		DEFDBL A-C
		i% = 0
		A = 1.5
		B = 2.5
		WHILE i% < 10000
		  i% = i% + 1
		  C = A / B + A * B
		WEND

Table 4.2 Source code for performance test.

Table 4.3 shows the execution times for the test. The threaded code executed roughly 5.6 times faster than MS-DOS Qbasic but 4 times slower than code compiled with Semantec C/C++. The code was executed on a 486/33 MHz PC.

	Execution Time (sec)
C/C++	0.11
TIL	0.44
Qbasic	2.47

Table 4.3 Test execution times.

The more significant improvements that would contribute to increases in execution speed are:

- Improve the design, algorithms, and data structures currently implemented in the translator.
- Code critical routines in assembler, in particular, the TIL inner interpreter routines and system primitives.
- Exploit the instruction set and architecture of the host machine to optimize routines, register and stack usage, and primitives.
- Increase the number of system primitives.

4.3 Language Extensions

4.3.1 User-Defined Extensions

No single system can be designed to satisfy all the requirements of all the possible users that will ever come in contact with it. To this end, user-defined language extensions are the key to a powerful and flexible system that allows a user to customize an environment particular to their needs. Libraries of user-defined macros can transform a general purpose system into a specialized computing environment. For example, the mechanical engineer might develop the following libraries:

- Stress analysis, kinematics and dynamics, heat transfer, fluid mechanics
- Specialized mathematics
- Engineering economics
- Databases; e.g., material properties, engineering and scientific constants, etc.
- Parts; parametrically driven geometrical parts database

When an application is eventually incorporated into the system, user-defined macros will provide a powerful mechanism to support and drive the application.

4.3.2 System Extensions

System extensions are absolutely necessary to expand the current language and incorporate an application command set at some future point in time. The translator was designed to be easily extensible by using the following language, translator, and development schemes:

- Context-free grammar specification
- Recursive descent parser with syntax-directed translation scheme
- C/C++ development language
- Modular design

4.4 Application Integration

Integrating an application into the system will bring to bear the full power of the translator as it serves as a core engine for the entire application. Presently, the system is suitable for integration into a wide variety of engineering applications. To this end, the following sections explore some possible design integration schemes.

4.4.1 Simulation Modeler

A simulation package is an excellent candidate for an application that can be incorporated into the system in a minimal amount of time. The familiar CSMP (Continuous System Modeling Program) developed by IBM is a good example of such a system. CSMP is itself an application-oriented programming language incorporating the FORTRAN language and an application-oriented command set tailored towards solving ordinary differential equations and block diagram simulations.

An actual CSMP program is shown in Figure 4.14. Programs are typically structured into three segments; *Initial*, *Dynamic*, and *Terminal*.[10] The Initial segment is primarily for defining constants, initial conditions for integration, and one-time calculations. The Dynamic segment contains all the simulation statements relative to solving the problem. This segment is iterative and is executed each time step during the integration. The Terminal segment contains any calculations or actions required after the simulation is completed such as plotting the results. Similar to FORTRAN subroutines, *Functional Blocks* are special CSMP functions. These functions consist of mathematical functions, system macros, switching functions, function generators, signal sources, logic functions, and FORTRAN functions.

Conceptually, CSMP can be viewed as a FORTRAN translator with application-oriented language extensions and a modified language syntax. In actuality, it's a front-end translator that takes as input CSMP statements and produces as output a FORTRAN source language subroutine. This subroutine is then compiled with an integration routine using a standard FORTRAN compiler and then executed.[11] Functional blocks would be supplied FORTRAN subroutines that come with the CSMP package.

Developing a similar system around the translator presented in this paper should be a relatively easy task once pointers are included in the language. The examples presented earlier for the Convolution Integral and Runge-Kutta DEQ solvers were themselves attempts by the user to create application-specific environments for solving simulation problems. User-written solvers and function generators were compiled into the language. Structured input files provided a programming environment tailored towards solving the simulation problem and removed the heavy burden of programming from the user. However, these examples were user-defined extensions to the language. A true application environment must incorporate system extensions into the language.

```
INITIAL
  CONSTANT W=10.0, C=.00259, A=-0.6, B=1.0, K1=.255, K2=1.025
  INCON X0=0.0, XDOT0=8.0
       M=W/386.
       COEF=C/M
DYNAMIC
  NOSORT
       IF(X.GE.0.) GO TO 2
       FOFX=K1*DEADSP(A,0.,X)
       GO TO 3
  2    FOFX=K2*DEADSP(0.,B,X)
  3    CONTINUE
  SORT
       XDDOT=-COEF*XDOT-FOFX/M
       XDOT=INTGRL(XDOT0,XDDOT)
       X=INTGRL(X0,XDOT)
TERMINAL
       KE=0.5*M*XDOT**2
       WRITE(6,4) KE
  4    FORMAT(' ',E16.7)
  TIMER DELT=.05, OUTDEL=0.2,FINTIM=60.
  METHOD RKSFX
  LABEL SPRING MASS VISCOUSLY DAMPED WITH DEADSPACE
  PRTPLOT XDOT,X,FOFX
  END
  STOP
ENDJOB
```

Figure 4.14 A typical CSMP program.

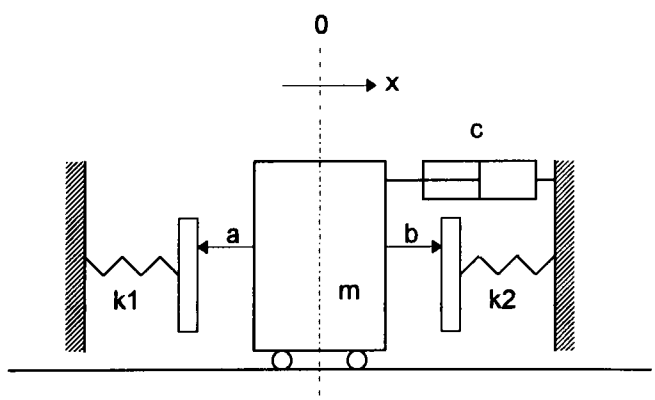


Figure 4.15 Viscously damped spring mass system.

The CSMP program shown in Figure 4.14 is taken from James, Smith, and Wolford.[12] The program models a viscously damped spring mass system as depicted in Figure 4.15. The mass slides on a frictionless surface while oscillating between two springs. A deadspace exists such that the mass is free to slide over a portion of its travel with no contact with either spring.

A possible translator scheme is presented below to implement a simulation package similar to CSMP. Much of the detail is left out for now, but an attempt is made to describe the major steps and translator modifications required to build a simulator package.

The proposed input deck for the simulator is shown in Figure 4.16.

```
CONTROL
METHOD RKSFX
TIMER DELT=0.05,OUTDEL=0.2,FINTIM=60.0
LABEL "Spring mass viscously damped with deadspace"
PRTPLOT xdot,x,fofx
INITIAL
FLOAT w,c,a,b,k1,k2,m,coef,fofx,ke,x0,xdot0
FLOAT x,xdot,xddot
w=10.0, c=0.00259, a=-0.6, b=1.0, k1=0.255, k2=1.025
x0=0.0, xdot0=8.0
m=w/386.0, coef=c/m
DYNAMIC
IF(x<0)
  fofx=k1*deadsp(a,0,x)
ELSE
  fofx=k2*deadsp(0,b,x)
ENDIF
xddot=-coef*xdot-fofx/m
xdot=INTGRL(xdot0,xddot)
x=INTGRL(x0,xdot)
TERMINAL
ke=0.5*m*xdot^2
PRINT ke
ENDJOB
```

Figure 4.16 Simulator input deck.

This scheme segments the input data file much the same as CSMP. The sections consist of the following:

Control	Execution and output control parameters such as integration step size, integration method, plot parameters, etc.
Initial	Defined constants, initial conditions for integration, and one-time calculations.
Dynamic	Iterative section to specify simulation statements relative to solving the problem.
Terminal	Calculations or actions required after the simulation is completed.

To support a simulation environment, the translator would have to be modified to support the following items:

System Variables	Global system variables accessible to the user and integration routines. **_METHOD, _TIME, _STARTIM, _FINTIM, _DELT, _LABEL**, etc
Simulation Statements	Extensions to the system grammar, parser routines, and runtime code to support a simulation command set. **CONTROL, METHOD** <*integration_id*>, **PRTPLOT** <*plot_list*>, **TERMINAL**, etc.
Simulation Functions	Extensions to the system grammar, parser routines, and runtime code to support a simulation function set. **INTGRL(), DERIV(), PULSE(), RAMP(), STEP(), DEADSP()**, etc.

Translation of the input deck would require the following actions to be produced by the translator:

1. Declaration and assignment of default values to global system variables.

   FLOAT _METHOD, _TIME, _STARTIM, _FINTIM, _DELT, _LABEL, _PRTPLOT, etc

   _METHOD=1, _STARTIM=0, etc

2. Translation of any user-defined functions created in the input deck. Any user-defined functions would precede the CONTROL block in the data deck.

3. Translation of CONTROL statements.

   ```
   METHOD RKSFX
   TIMER DELT=0.05,OUTDEL=0.2,FINTIM=60.0
   LABEL "Spring mass viscously damped with deadspace"
   PRTPLOT xdot,x,fofx
   ```

4. Translation of INITIAL, DYNAMIC, and TERMINAL statements. The translator will actually build a function called update() as shown below, embed appropriate statements as required, and then compile the function.

```
DEFINE update()
    EXTERN FLOAT _TIME, _STARTIM, _FINTIM, _DELT, etc
    FLOAT w,c,a,b,k1,k2,m,coef,fofx,ke,x0,xdot0
    FLOAT x,xdot,xddot
    w=10.0, c=0.00259, a=-0.6, b=1.0, k1=0.255, k2=1.025
    x0=0.0, xdot0=8.0
    m=w/386.0, coef=c/m
    FOR(_TIME=_STARTIM; _TIME<=FINTIM; _TIME=_TIME+DELT)
        IF(x<0)
            fofx=k1*DEADSP(a,0,x)
        ELSE
            fofx=k2*DEADSP(0,b,x)
        ENDIF
            xddot=-coef*xdot-fofx/m
        xdot=INTGRL(xdot0,xddot)
        x=INTGRL(x0,xdot)
        WRITE()
    NEXT
    PLOT()
    ke=0.5*m*xdot^2
    PRINT ke
END_DEFINE
```

5. Generate a function call to execute the simulation: **update()**

6. Generate a call to exit the translator and end the simulator run: **exit**

The translation steps presented above are summarized below:

1. Declaration and assignment of global system variables.
2. Translation of user-defined functions.
3. Translation of Control statements.
4. Parse Initial, Dynamic, and Terminal statements; construct and compile simulation routine update().
5. Execute update() to run the simulation.
6. Exit the translator to end the simulation run.

4.4.2 Finite Element Analysis

Building a Finite Element Analysis package with interactive pre and post processing is a very large project. A truly professional package must provide support for the following:

- Interactive graphics with a modern user interface including menus, forms, and command line input.
- A solid modeler for geometry construction.
- Robust element library.
- Meshers.
- Material, property, load, and boundary condition specification.
- Numerical solvers.
- Post-processing support for viewing, manipulation, and plotting of results.
- A command language to interface with the system.

No attempt will be made here to look at specific translator modifications to build an FEA application. Instead, a general outline for an FEA system will be presented where the translator serves as a central engine to drive the entire system as shown in Figure 4.17.

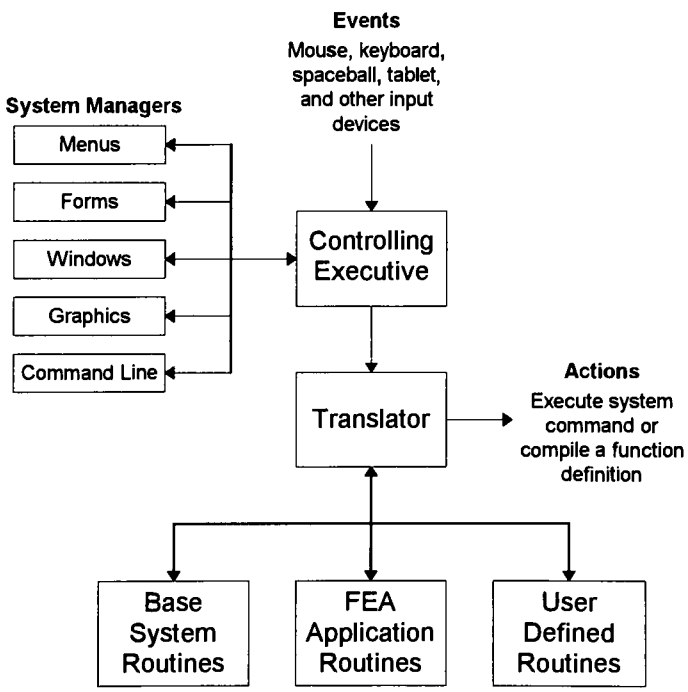


Figure 4.17 Translator as a central engine for an FEA system.

One possible design to build a graphics-oriented, interactive software system around the translator can be accomplished using the following:

- Controlling Executive
- System Managers; menus, forms, windows, graphics, and command line
- Translator
- Application Routines

The *Controlling Executive* serves the purpose of overseeing all system events and executing the appropriate routines to manage those events. After system startup and initialization, program control is passed to the controlling executive. The controlling executive will then monitor events, pass program control to system managers based on the type of event and current status of the managers, acknowledge and act upon registered requests from system managers, and execute appropriate routines or pass control to the translator. Program control always returns to the controlling executive.

System Managers are specialized routines designed to manage the different areas of the Graphical User Interface (GUI). A modern GUI must support and manage menus, forms, command line input, graphics windows, etc. Details aside, managers serve the function of processing user inputs and generating actions based on those inputs. Managers work with the controlling executive and amongst themselves to effectively service the application.

The *Translator* is the core engine that generates the actions to drive the application. Almost all application actions occur by processing a command stream thru the translator. The system language will have specific extensions to access most all of the applications routines. In addition, extensions to access system routines will be highly desirable for the user to customize menus, forms, etc.

Application Routines encompass all the routines specific to the application. For an FEA system, typical application routines would be solvers, mesh generators, solid geometry modelers, post processing routines, etc.

As an example, assume system startup and initialization has occurred and program control is passed to the controlling executive. The controlling executive will then go into a loop polling for an event such as a keystroke or mouse movement. If the event is a keystroke and no manager is active, control is passed to the command line manager to service the event. The command line manager would register itself as active, fetch the keystroke, and copy the character to command stream buffer. Control would then pass back to the controlling executive to resume event polling. The next keystroke would trigger the same process but in this case control is passed to the command line manager because it is the active manager. The concept of active manager is necessary because, say for example, the forms manager is active and the user is typing in an entry on the form. The controlling executive must in this case pass control to the forms manager when a keystroke event occurs, not the command line manager. The forms manager would then call the command line manager to process the keystroke. Anyways, assume the process continues until <ENTER> is pressed on keyboard. At this point, the command line manager recognizes that input from the keyboard is finished. The command line manager will then null terminate the command stream buffer, register itself as inactive, register a request to call the translator, and return control to the controlling executive. The controlling executive will then pass control to the translator. At this point, the translator processes the input command stream.

In the example above, if the event had been a mouse click on a menu, control would have been passed to the menu manager. The menu manager would then determine which menu item was clicked, copy the command string associated with that item to the command stream buffer, register a request to call the translator, and return control to the controlling executive. The controlling executive would then pass control to the translator. At this point, the translator processes the input command stream.

Many different designs are plausible and the brief description given here is by no means the best. The main point to this design is the central role the translator would serve. By driving an application thru the translator, the user will be given access to almost the entire application command set, system routines, and the base translator language. The flexibility and power to drive the application is greatly enhanced for the user and to this end, so is the ability to solve the most demanding engineering problems faced today. Software sales probably won't hurt either!

5. RESULTS

Efforts put into the project resulted in the following major achievements:

- Complete theory and design details for all phases of the translator and language.
- *CP.EXE* - An executable translator program that creates a programming and command interpreter environment. The program includes an integrated application shell to provide the user with an interactive command line environment.
- A complete Language and Compiler Guide.
- A set of diagnostic tools to aid the developer.
- Example code solving engineering problems; Four-Bar Crank Mechanism, Material Library, Integration Solver, Runge-Kutta DEQ Solver, and Convolution Integral.
- Performance evaluation including a comparison against C/C++ and MS-DOS QBasic.
- Exploration in application integration for a simulation package similar to CSMP.

Translator

The translator developed for the project is both an interpreter and compiler. Input is in the form of a *command stream* which is composed of two types of objects; command statements and function definitions. The translator correspondingly has two modes of operation; a command or interpreter mode that executes command statements, and a compile mode that compiles functions into threaded code to become new commands in the system. See Figure 5.1.

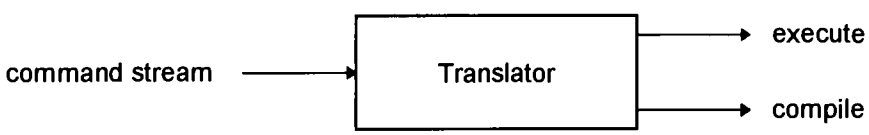


Figure 5.1 Basic translator operation.

A *command statement* is any valid statement that appears outside of a function definition; e.g., declaration, assignment, function call, etc. A *function definition* is a collection of valid statements that define and make up a function. Function definitions start with the keyword DEFINE and end with the keyword END_DEFINE. Command statements and function definitions together comprise the language of the translator. A typical command stream is shown in Figure 5.2.

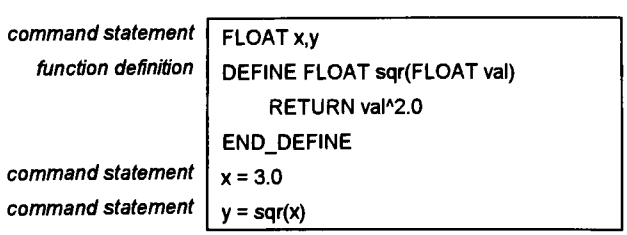


Figure 5.2 Typical command stream.

The translator is actually comprised of two subsystems; a front end translator incorporating a recursive descent parser and a back end translator consisting of a threaded code interpreter very similar to a FORTH interpreter. See Figure 5.3. The front end handles the tasks of lexical, syntax, and semantic analysis, type checking, intermediate code generation, and symbol table management. The back end consists of a threaded code interpreter and compiler. The threaded interpreter consists of a keyword dictionary for storing system primitives and user defined secondaries, stacks, registers, and a software interpreter routine that mechanizes the language.

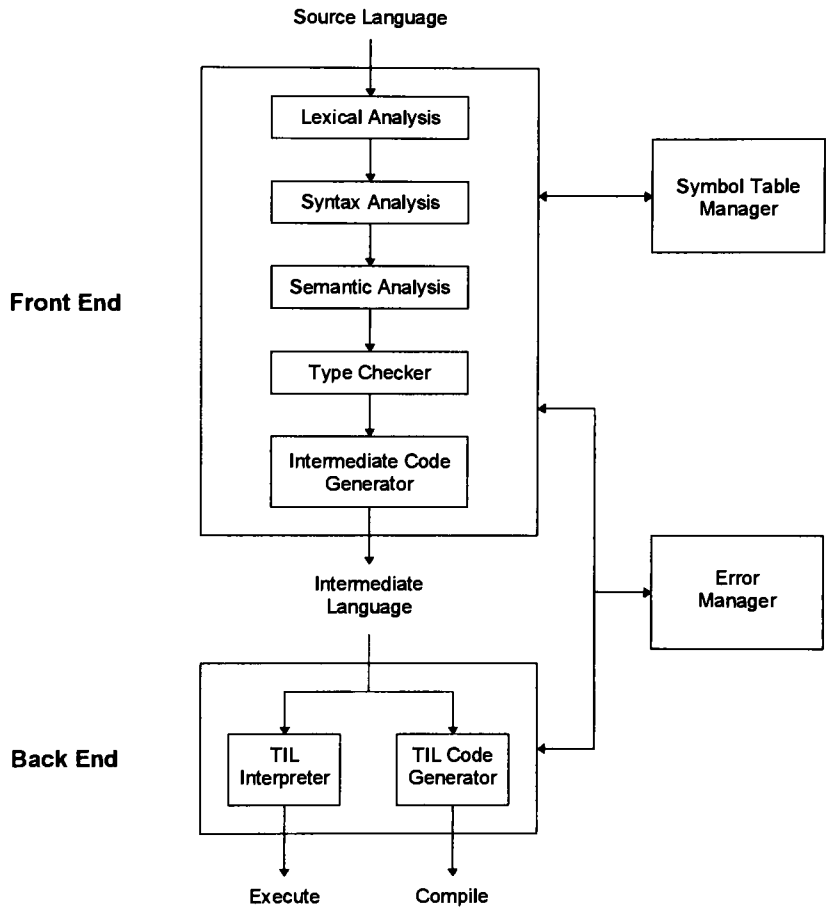


Figure 5.3 Translator design.

The front end parser and back end threaded code interpreter are nearly independent of one another. In fact, during the early development stages of the project, a threaded code interpreter was developed first. This resulted in a translator which operated very much the same way as a programmable RPN calculator. This type of system uses a stack for storing operands and results. Operands are first pushed onto the stack, then the operation is performed. Calculations using RPN is very efficient and preferred by many engineers. However, writing all but the simplest programs in RPN is quite cumbersome, much the same as coding in assembler. To serve as a core engine for an application, a modern procedural language similar to FORTRAN or BASIC is much more desirable. To accomplish this, a front end translator was designed and

constructed to process a high-level procedural language. To utilize the existing threaded code interpreter, an RPN intermediate language form was chosen. Early work on the front end translator was independent of the back end. For a period of time during development, the front end translator and back end threaded code interpreter existed as two separate programs. Output from the front end was piped to a file which in turn became input for the back end. Eventually the two systems were physically integrated into one program, but the design and source code modules remain distinct. As a development tool, current provisions in the translator allow the developer to view intermediate language generation, threaded code generation, and totally bypass the front end and work directly with the back end threaded interpreter.

The main features of the translator are listed below.

The *front end* translator:

- Takes as input a fully structured source language very similar to BASIC or FORTRAN.
- A top down recursive descent parsing technique is used to implement a syntax-directed translation scheme for infix to postfix notation with embedded semantic actions.
- Type checking and implicit casting are integrated into the parser.
- A separate lexical analyzer handles the task of token construction, classification, and attribute binding.
- All storage is static; symbol table management and memory allocation tasks are handled by the front end.
- Extensive error checking is performed during all phases of translation.
- Produces as output an intermediate language in postfix notation, type checked, and syntactically correct in form.

The *back end* threaded interpreter/compiler:

- A threaded code interpreter very similar to a FORTH compiler that takes as input a postfix language consisting of numbers, addresses, and keywords.
- Two modes of operation; an interpreter mode and a compile mode. In compile mode, keyword definitions are compiled into threaded code to become new keywords in the language. In interpreter mode, keywords are executed immediately.
- A software interpreter very similar to the actual hardware interpreter in a computer is mechanized to execute the threaded code.
- Floating point exceptions and runtime array bounds checking are supported.

Ease of extensibility for development and application integration was a primary concern in the design of the translator. Major design features implemented to accomplish this were:

- Context-free grammar.
- Syntax directed translation scheme.
- Recursive descent parser.
- Modular design of translator.
- Portable to other hardware platforms; the entire system was written in C/C++.

Language

A high-level procedural language similar to FORTRAN or BASIC was integrated into the translator. The language supports modern programming constructs such as user-defined types or records, variables, multi-dimensional arrays, functions, and program flow control statements. The ability to write functions or macros allows the user to extend the language command set, customize the environment, develop specialized libraries of routines, and write programs to drive an application. For instance, a collection of routines to calculate beam deflections can be written and stored to disk. When the user needs to do this type of analysis, simply loading the macro file will transform the system into a specialized computing environment for analyzing beam problems. Additional routines can be added as needed. The Material Library example presented earlier in this paper adds a material database to the system. Once loaded, material properties can be used in calculations, passed as parameters to other routines, etc. When an application is incorporated into the system, user-defined macros will provide a powerful mechanism to support and drive the application. Macros can be loaded at the beginning of a session to customize an environment tailored to the user's needs. Menu layouts, viewport configurations, default settings, and other items can be configured at system startup. For a system such as an FEA modeler, macros can be developed to generate geometry, define complex load and boundary conditions, maintain a material and property database, control multiple runs, post process results, etc.

Additions to the language and modification of source code is a relatively easy task once the developer familiarizes themselves with the grammar, source code, and program structure. Simple constructs can be added to the system in minutes, however, more complex additions may involve hours for design and integration. For example, to add a construct to convert an integer to an ascii string would probably take under 10 minutes in total time because a library function in C exists to do this. If an FEA application is being written, adding a construct to call a mesher routine would probably involve 30 minutes or so assuming the mesher routine exists. At the other extreme, adding pointer variables to the existing system language would probably involve weeks of work.

The major features of the language include:

- Free format of source language text.
- Symbolic constants.
- Variables and multidimensional arrays.
- User-defined types or records.
- User-defined functions that take parameters and return values.
- Program flow control constructs; e.g., IF, SWITCH, FOR, REPEAT, WHILE, etc.
- Local, global, and external objects.
- Mathematical functions; e.g., SIN(x), ABS(x), COSH(x), LN(x), etc.
- Complex arithmetic expressions; e.g., x = EXP(-zeta*wn*(t-s))*SIN(SQRT(1.0-zeta^2.0)*wn*(t-s)).
- Screen and disk I/O functions; e.g., OPEN, CLOSE, PRINT, PRINT #, INPUT, INPUT #, etc.

The reader is referred to the Language and Compiler guide included in the Appendix for a complete description of the language.

Major language constructs are summarized in Table 5.1; operators, precedence, and associativity in Table 5.2; and the entire language token set in Table 5.3. A complete grammar specification for the entire language is given in Section 3.7.

CONSTRUCT	EXAMPLE/DESCRIPTION
Base Types	FLOAT, INTEGER, and STRING
User-Defined Types or Records	TYPEDEF POINT {FLOAT mag, FLOAT ang} POINT r r.mag = 5.5, r.ang = 30.0
Variables and Multi-Dimensional Arrays	FLOAT vector, cam[-2:3,3] vector = 4.0, cam[-1,2] = vector
Functions	DEFINE FLOAT sqr(FLOAT val) RETURN val^2 END_DEFINE x = sqr(5)
Conditional Statements	IF...ELSEIF...ELSE...ENDIF SWITCH...CASE...DEFAULT...ENDSWITCH
Iterative Statements	FOR...NEXT WHILE...ENDWHILE REPEAT...UNTIL
Symbolic Constants	SYMBOL pi "3.141592654" x = pi/180
Mathematical Functions	SIN(x), ABS(x), COSH(x), LN(x), etc.
I/O	OPEN, CLOSE, INPUT, PRINT, LOAD

Table 5.1 Language constructs.

Operator	Operation	Associativity
( ) [ ] { } : .	Expression	L>R
+ - NOT	Unary	L>R
^	Exponentiation	R>L
* /	Multiplicative	L>R
+ -	Additive	L>R
< <= > >=	Relational	L>R
= <>	Equality	L>R
AND	Logical	L>R
OR	Logical	L>R
=	Assignment	R>L
,	Sequencing	L>R

Table 5.2 Operators, precedence, and associativity.

ε	.	,	;	:
&	(	)	[	]
{	}	=	<>	<
<=	>	>=	+	-
*	/	^	OR	AND
NOT	LET	TYPEDEF	DEFINE	END_DEFINE
RETURN	FOR	NEXT	GLOBAL	LOCAL
IF	ELSEIF	ELSE	ENDIF	EXTERN
SWITCH	CASE	DEFAULT	ENDSWITCH	BREAK
WHILE	ENDWHILE	REPEAT	UNTIL	TRANSLATE
FLOAT	INTEGER	STRING	INPUT	PRINT
OPEN	CLOSE	LOAD	FTOA	DELETE
SYMBOL	SYSTEM	SIN	COS	TAN
ASIN	ACOS	ATAN	SINH	COSH
TANH	ABS	SQRT	LOG	LN
EXP	*identifier*	*string*	*unsigned real*	*unsigned integer*
user defined type				

Table 5.3 Language token set.

CP.EXE Program

An application shell was written to provide the user with an interactive command line environment for using the translator. The shell and translator comprise a single executable program: *cp.exe*. After system startup and initialization, a startup screen appears as shown in Figure 5.4. A title banner is displayed showing the current version and release date of the system. The command line prompt is displayed and the system is ready for user input. At this point, a programming and command interpreter environment is created that lasts for the duration of the application session. Objects such as variables, arrays, symbolic constants, and user-defined functions can be created. Once created, objects exist for the entire application session and can be used in expressions, as parameters for commands, or as functions to perform calculations or drive the application. Actions are generated by processing commands thru the translator. The user generates commands by entering input at the keyboard or loading prewritten macro files from disk.

The Language and Compiler guide included in the Appendix contains a complete operations guide and tutorial on how to run the translator program cp.exe.

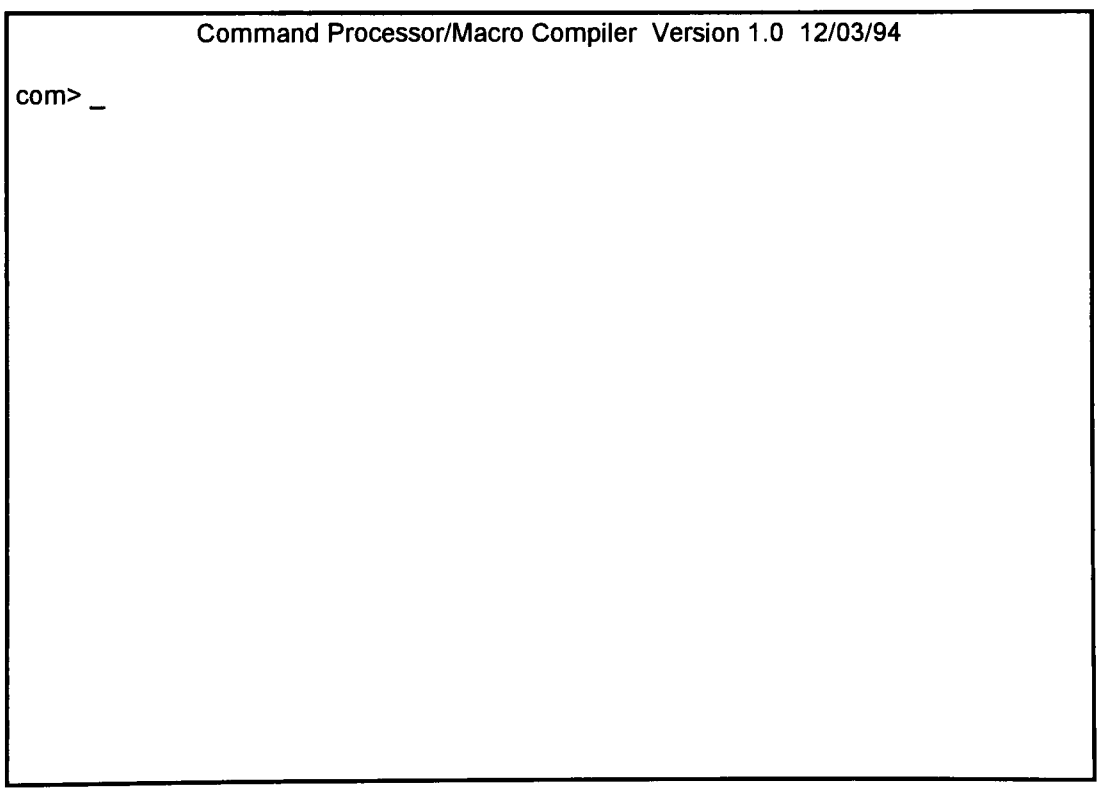


Figure 5.4 Translator startup screen.

Language and Compiler Guide

A complete Language and Compiler Guide was written and included in the Appendix. This guide describes the entire language in detail and presents several examples. An operations guide and tutorial is included to instruct the reader on how to run the program and also describes the use of the diagnostic tools included for the developer. A complete list of system error messages is also included.

Diagnostic Tools

To help aid the application developer, a set of diagnostics tools was incorporated into the translator design. These tools allow the developer to:

- Display the contents of the symbol table.
- Display intermediate language output generated by the front end parser.
- Display compiled threaded code generated by the back end compiler.
- Bypass the front end parser and work directly with the back end threaded interpreter/compiler.
- Display destructor actions of symbol table objects.

Error Management

Error management was given special emphasis from the very start of the project, right down to the choice of compilers for development. The Zortech C++ compiler included the latest extensions under development by NCEG, the Numerical C Extensions Group. These extensions provide support for floating point entities such as infinity, +0 and -0, and nans. Infinity results when 1/0 is evaluated. Unless special precautions are taken, most programs crash under this condition. Implementing NCEG extensions, the result of this operation yields INFINITY which is a nan (not a number.) Nans can be carried thru calculations just like any other floating point number except the result is always a nan. This allows a routine to carry out a numerical algorithm to completion and afterwards check for the presence of a nan in the result. This would indicate an error has occurred and appropriate action could be implemented to handle it.

Comprehensive error checking during all phases of translation and runtime execution was designed into the translator. An error management module services all the system routines. When an error occurs, translation stops and the translator is reset to recover from the error. An error message is displayed to the user including the source text line containing the error and the offending tokens are highlighted. If the source text originated from a file, the file name and line number are also displayed. Error management currently stops translation at the first occurrence of an error. However, the system is designed to handle multiple error registrations if future translator development provides for continued translation after one or more errors has occurred. Most compilers incorporate this feature and even allow the user to specify how many errors can occur before translation terminates.

Engineering Application

The paper presented several engineering examples to illustrate the current capabilities of the translator; a Four-Bar Crank Mechanism, Material Library, Integration Solver, Fourth-Order Runge-Kutta DEQ Solver, and Convolution Integral. The examples clearly demonstrated the programming capabilities of the current language and performance of the translator. In addition, the reentrant capabilities of the translator were exploited using the TRANSLATE() command. At runtime, a call to TRANSLATE() will suspend the current operation of the back end interpreter, save the register contents, preserve the stack contents, and then reenter the translator to process a command stream passed as a string variable. After the stream is processed, control returns back to the routine that issued the call to TRANSLATE() and continues execution of that routine. Basically, this feature allows a running program to suspend itself, reenter the translator to process commands and compile new functions into the system, and then return and continue program execution. This capability allows the user to specify constants, variables, and functions in a data file and let the program handle the job of compiling the objects into the system. For example, the Runge-Kutta DEQ Solver presented earlier in this paper is designed to read an input data file which contains the differential equations to be solved. This eliminates the user's burden of having to write and compile functions each time a new set of differential equations is analyzed as is typical with FORTRAN or BASIC. The user only modifies the data file, not the program code. To my knowledge, this reentrant capability is unique to this translator, and at present, its implications have not yet been explored in much detail.

The following engineering examples were presented to explore the capabilities of the translator.

1. Four-Bar Crank Mechanism - Solution to output angles for a four-bar crank mechanism. The solution involved solving Freudenstein's equation using the interative Newton-Raphson method. The purpose of this example was primarily to illustrate the source language and writing a function.

2. Material Library - Adding a material library into an application environment. Example presents a user-written macro that maintains a material database and includes a routine for adding additional materials during an application session.

3. Integration Solver - Integration routine using the Trapezoidal method. Example presents a user-written macro that adds an integration function to the language. What is unique about this routine is the fact that any single valued, continuous function can be passed to the integration routine as a string variable. The integration routine will then compile the function into the system, perform the integration, delete the function, and then return the result. For example, to perform the integration $\int x^2+3\ dx$ from x= 1 to x =3 using 100 steps and print the result to the screen, all the user has to do is type the following at the command line:

   com> PRINT INTEGRATE(1,3,100,"x^2+3") <ENTER>

   The result of the integration is then printed to the screen. This example demonstrates the reentrant capabilities of the translator using the TRANSLATE() command.

4. Fourth-Order Runge-Kutta DEQ Solver - General routine to solve a system of first order differential equations using the fourth-order Runge-Kutta method. Example presents a user-written macro to solve a system of first-order differential equations, in particular, the solution to the flight of a rocket with variable weight is analyzed. This routine uses an input data file for complete specification of the entire problem. The user only prepares an input data file for each problem, no code has to be written. Constants, functions, and differential equations are specified in the data file; the solver routine handles the task of compiling the objects into the system using the TRANSLATE() command.

5. Convolution Integral - General routine to solve for the response of a system using convolution. Example presents a user-written macro to solve for the response of a system using convolution, in particular, the solution to an underdamped, second order system subjected to a pulse generator forcing function is analyzed. This routine uses an input data file for complete specification of the entire problem. The user only prepares an input data file for each problem, no code has to be written. Constants, unit impulse response, and piecewise forcing functions are specified in the data file; the solver routine handles the task of compiling the objects into the system using the TRANSLATE() command.

Performance

Overall, translator performance was very acceptable considering the fact that the translator was written entirely in C/C++ and designed to be portable to different platforms. Compile times for numerical routines such as the Runge-Kutta DEQ solver or Convolution Integral presented earlier in this paper were under 0.2 sec. Approximate compilation times for the examples presented earlier are tabulated in Table 4.1. The files were compiled on a 486/33 MHz PC.

Macro File Name	Compilation Time (sec)
4bar.mac	0.11
matlib.mac	0.17
integrat.mac	0.05
rkutta.mac	0.11
convolut.mac	0.16

Table 5.1 Compilation times for a few selected macro files.

Execution speed was also good. A simple test program to perform floating point arithmetic was developed and run using C/C++, TIL, and MS-DOS Qbasic. Table 4.2 lists the source code for the test.

C/C++ Source Code	Translator Source Code	Qbasic Source Code

void main()
{
int i;
double a,b,c;
i=0;
a=1.5,b=2.5;
while(i<10000)
{
i=i+1;
c=a/b+a*b;
}
}
	DEFINE test()
	INTEGER i
	FLOAT a,b,c
	i=0
	a=1.5,b=2.5
	WHILE(i<10000)
	i=i+1
	c=a/b+a*b
	ENDWHILE
	END_DEFINE
		DEFDBL A-C
		i% = 0
		A = 1.5
		B = 2.5
		WHILE i% < 10000
		i% = i% + 1
		C = A / B + A * B
		WEND

Table 5.2 Source code for performance test.

Table 4.3 shows the execution times for the test. The threaded code executed roughly 5.6 times faster than MS-DOS Qbasic but 4 times slower than code compiled with Semantec C/C++. The code was executed on a 486/33 MHz PC.

	Execution Time (sec)
C/C++	0.11
TIL	0.44
Qbasic	2.47

Table 5.3 Test execution times.

Issues surrounding translator performance were covered with respect to compilation and execution. Compilation speed is mainly dependent on the design and efficiency of the parsing method, search algorithms, memory allocation routines, stack and register usage, and dictionary management routines. The recursive-descent parser currently implemented in the translator relies heavily on recursive function calls which are time consuming processes. Redesigning the parser to implement a bottom up method could substantially increase performance but would require a tremendous effort. The current symbol table and TIL dictionary routines use linear search algorithms to lookup objects. Implementing a hashing scheme would significantly improve search times when the application environment becomes large.

Execution speed is mainly dependent on the design and efficiency of the threaded interpreter algorithms, stacks and register usage, and primitive source code algorithms. The threaded interpreter routines and primitives are critical to execution speed. These routines comprise the instruction set and software interpreter that make up and mechanize the language. Every effort must be made to maximize the speed and efficiency of these routines. If a particular target machine is to be used exclusively, it would be well worth the effort to recode these routines in assembler to maximize performance.

Application Integration

Application integration was investigated for a simulation package similar to CSMP and is presented in Section 4.4.1. Developing a similar system around the translator presented in this paper should be a relatively easy task once pointers are included in the language. Translator modifications to develop a simulation package would require the addition of global system variables and extensions to the system grammar, parser routines, and runtime code to support a simulation command set. Excluding postprocessing routines, a basic simulator system could probably be developed in a month or so. Equivalent input decks for a CSMP program and a proposed simulator program are shown in Table 5.4.

CSMP INPUT DECK	SIMULATOR INPUT DECK

INITIAL
  CONSTANT W=10.0, C=.00259, A=-0.6, B=1.0, K1=.255, K2=1.025
  INCON X0=0.0, XDOT0=8.0
  M=W/386.
  COEF=C/M
DYNAMIC
  NOSORT
  IF(X.GE.0.) GO TO 2
  FOFX=K1*DEADSP(A,0.,X)
  GO TO 3
  2 FOFX=K2*DEADSP(0.,B,X)
  3 CONTINUE
  SORT
  XDDOT=-COEF*XDOT-FOFX/M
  XDOT=INTGRL(XDOT0,XDDOT)
  X=INTGRL(X0,XDOT)
TERMINAL
  KE=0.5*M*XDOT**2
  WRITE(6,4) KE
  4 FORMAT(' ',E16.7)
  TIMER DELT=.05, OUTDEL=0.2,FINTIM=60.
  METHOD RKSFX
  LABEL SPRING MASS VISCOUSLY DAMPED WITH DEADSPACE
  PRTPLOT XDOT,X,FOFX
  END
  STOP
ENDJOB
	CONTROL
	  METHOD RKSFX
	  TIMER DELT=0.05,OUTDEL=0.2,FINTIM=60.0
	  LABEL "Spring mass viscously damped with deadspace"
	  PRTPLOT xdot,x,fofx
	INITIAL
	  FLOAT w,c,a,b,k1,k2,m,coef,fofx,ke,x0,xdot0
	  FLOAT x,xdot,xddot
	  w=10.0, c=0.00259, a=-0.6, b=1.0, k1=0.255, k2=1.025
	  x0=0.0, xdot0=8.0
	  m=w/386.0, coef=c/m
	DYNAMIC
	  IF(x<0)
	  fofx=k1*deadsp(a,0,x)
	  ELSE
	  fofx=k2*deadsp(0,b,x)
	  ENDIF
	  xddot=-coef*xdot-fofx/m
	  xdot=INTGRL(xdot0,xddot)
	  x=INTGRL(x0,xdot)
	TERMINAL
	  ke=0.5*m*xdot^2
	  PRINT ke
	ENDJOB

Table 5.4 Equivalent simulator input decks.

A general discussion on integrating an FEA application was also presented in Section 4.4.2.

6. CONCLUSIONS

The translator, language, and application shell developed for the project should provide the developer with a solid foundation and framework to integrate an engineering application. The translator is modular, easily extensible, and written entirely in C/C++. The actual development of the translator and language was an evolutionary process and is testament to the ease of extensibility designed into the system. Portability to another platform other than DOS should be quite easy. The diagnostic tools integrated into the translator are a valuable asset to the developer and should assist greatly in future development efforts. Intermediate and threaded code display proved to be very important tools for debugging front and back end translator routines during development. The current language gives the user a high-level procedural language capable of solving very complex engineering problems quite readily. When an application is integrated, the language will provide the user with a powerful means to interface with and drive the application.

In its present form, the translator, language, and command line shell can be used for several purposes:

- Use it like a FORTRAN or BASIC compiler to compile and run numerical programs.
- Use it as an interactive computing environment to perform specialized computations. Create libraries of macros to perform special calculations such as stress analysis, kinematics and dynamics, heat transfer, fluid mechanics, advanced mathematics, and engineering economics.
- Use it to develop and maintain databases of materials and other objects.
- Use it as an application-specific solver as was done for the Runge-Kutta DEQ and Convolution Integral solvers presented earlier in this paper. Supporting programs can be developed in C/C++, FORTRAN, or BASIC to add additional capabilities to the system such as postprocessing and graphical display of results.

Ultimately, an entire application can be developed and integrated into the language, translator, and application shell.

As with all projects, time and other constraints force compromises in design. The translator and language both have room for improvements; some recommendations for future additions are summarized below:

Translator:

- Add dynamic memory allocation. Currently, all storage is static and allocated at compile time. This is advantageous when an algorithm repetitively calls a function, but seldom called routines would be better off using dynamic storage and releasing memory when not being used.
- Add a string space manager to the system. String literals are currently lost on heap and memory is not released. For example, s = "Hello " + "World" assigns "Hello World" to s, but the string literals "Hello " and "World" are lost on the heap.
- Add code optimization to the system.
- Add dynamic memory allocation for TIL dictionary entries. Currently a large block of fixed size is allocated at system startup.

- Implement C/C++ stream I/O.
- Integrate a debugger.

Language:

- Initialization during declaration; e.g., FLOAT mag = 5.0.
- Right to left assignment; e.g., LET a=b=c=3.14.
- Base type CHAR and operators to support character manipulation.
- Addition of pointers, especially pointers to functions for use with the TRANSLATE() command.
- Expand operator set for manipulating strings.
- Expand operations allowed on user defined types.
- Base type COMPLEX and operators to support it.
- Explicit cast functions such as FTOI(); float to integer conversion.
- Addition of CONTINUE statement for iterative flow constructs.
- Objections aside, addition of GOTO and LABEL statements.
- Permit objects local to a function to have the same name as a symbolic constant.

As a final note, much of the early work on the project was due to inspiration from a single source written by R. G. Loeliger, "*Threaded Interpretive Languages*". Later work on the front end translator was greatly influenced by Aho ,Sethi, and Ullman, "*Compilers: Principles, Techniques, and Tools*". These books are highly recommended to anyone wishing to learn about translators, and especially to anyone who might be interested in developing this project further.

7. REFERENCES

[1] P. M. Lewis II, D. J. Rosenkrantz, and R. E. Stearns, Compiler Design Theory (Reading, Mass.: Addison-Wesley, 1976), pp. 135-143.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, Compilers: Principles, Techniques, and Tools (Reading, Mass.: Addison-Wesley, 1986), pp. 176-178.

[3] Ibid., pp. 279-342.

[4] Ibid., pp. 585-722.

[5] Ibid., pp. 513-584.

[6] M. L. James, G. M. Smith, and J. C. Wolford, Applied Numerical Methods For Digital Computation (New York, NY: Harper & Row, Publishers, Inc., 1985), pp. 94-105.

[7] Richard L. Burden and J. Douglas Faires, Numerical Analysis (Boston, MA: Prindle, Weber, and Schmidt, 1985), pp. 261-265.

[8] James, Smith, and Wolford, pp. 479-487.

[9] R. G. Loeliger, Threaded Interpretive Languages (Peterborough, NH: Byte Publications, Inc., 1981), p. 6.

[10] Frank H. Speckhart and Walter L. Green, A Guide To Using CSMP - The Continuous System Modeling Program (Englewood Cliffs, NJ: Prentice-Hall, Inc., 1976), p. 6.

[11] James, Smith, and Wolford, p. 631.

[12] Ibid., pp. 631-635.

APPENDIX A: ZORTECH/SYMANTEC C++ COMPILER

The Zortech C/C++ Science and Engineering Edition was chosen as the development system for this project. The system was clearly superior to Microsoft and Borland C/C++ products at the time it was selected. The Zortech C/C++ system included the following:

- The Zortech C++ Workbench; a DOS based program editor and productivity tool.
- Royalty free 32 bit DOSX extender to provide a 4Gb flat address memory model.
- Flash Graphics Library that supplies graphics routines and video support for SVGA, VESA, 8514a, and TIGA.
- M++ Library that supplies classes and functions to create and manipulate arrays and matrices, perform linear algebra, eigensystem analysis, and more.
- IEEE and NECG compliant.
- Support for interfacing to FORTRAN routines.
- A C++ toolkit featuring link lists, queues and stacks, hashed search tables, etc.
- A C++ video course was available for purchase from Zortech.

During the development of this project, the Zortech C/C++ system became obsolete and no longer commercially available. The Symantec C++ Professional system replaced Zortech and the project was finished using the new system. Flash Graphics and M++ libraries are not bundled with the new system, but they are still available commercially.

APPENDIX B: SOURCE CODE MODULES

The entire project was written in C/C++. Source code modules for the translator are supplied on disk and include the following:

Header files:

cp.h, compiler.h, scanner.h, til.h, symtable.h, error.h, buffer.h, coleman.h

Source files:

cp.cpp	Program main() and application shell.
rdparser.cpp	Recursive-descent parser routines.
scanner.cpp	Lexical analysis routines.
til.cpp	Threaded code interpreter routines.
symtable.cpp	General purpose symbol table routines.
typetbl.cpp	Symbol table routines for base and user-defined types.
vartbl.cpp	Symbol table routines for variables.
arraytbl.cpp	Symbol table routines for arrays.
functbl.cpp	Symbol table routines for functions.
resrvtbl.cpp	Symbol table routines for reserved words.
deftbl.cpp	Symbol table routines for symbolic constants.
error.cpp	Error management routines.
buffer.cpp	Buffer management routines.
coleman.cpp	Miscellaneous general purpose routines.

A *makefile* is also supplied for use with the Symantec C/C++ compiler. In addition, the file *opt.bat* is included. This is a small batch file that simply executes the Symantec compiler *sc.exe* and passes the appropriate file names and command switches.

APPENDIX C: LANGUAGE AND COMPILER GUIDE

C1. Introduction

C1.1 The Command Processor

Since the command processor was developed to serve as an engine for an application, there is no explicit structure that defines a *program* in the usual sense. With this in mind, a better way to conceptualize the system is to view it as a processor that consumes *command streams* and produces *actions*.

A *command stream* is composed of two types of objects; command statements and function definitions. The processor correspondingly has two modes of operation; a command or interpreter mode that executes command statements, and a compile mode that compiles functions into new commands. See Figure C.1.

Figure C.1: Command processor.

A *command statement* is any valid statement that appears outside of a function definition; e.g., declaration, assignment, system command, function call, etc.

A *function definition* is a collection of valid statements that define and make up a function. Function definitions start with the keyword DEFINE and end with the keyword END_DEFINE.

For example, consider the command stream shown in Figure C.2.

```
FLOAT x,y

DEFINE FLOAT sqr(FLOAT x)
  RETURN x^2.0
END_DEFINE

LET x = 3.0
LET y = sqr(x)
```

Figure C.2: Typical command stream.

Processing of the stream objects and resulting actions is described in detail below.

FLOAT x,y
Object: Declaration statement outside of a function definition. The statement is processed in command or interpreter mode and executed.
Action: Two external floating point variables are created, x and y.

DEFINE FLOAT sqr(FLOAT x)
RETURN x^2.0
END_DEFINE
Object: A function definition. The function definition is processed in compile mode.
Action: The function sqr is compiled into the system but not executed.

LET x = 3.0
Object: Assignment statement outside of a function definition. The statement is processed in command or interpreter mode and executed.
Action: External floating point variable x is assigned value of 3.0.

LET y = sqr(x)
Object: Assignment statement outside of a function definition. The statement is processed in command or interpreter mode and executed.
Action: External floating point variable y is assigned return value of 9.0 from function call to sqr(x).

Although the previous example was very simple, it should clarify the dual role of the processor as an interpreter and compiler in an application environment.

Command streams are source independent as far as the processor is concerned. Central to an application, the processor would service inputs from:
- Menus, forms, and screen managers
- Command line input
- Disk files

Actions produced would range from changing a simple system setting to:
- Customizing the startup environment; settings, menus, viewports, etc.
- Loading and compiling libraries of special functions into the system
- Processing a session file and recreating a previous application session.
- Running a complex user defined program

C1.2 Uses and Limitations

At present, the full potential of the processor cannot be realized because no application has been integrated. However, the current system does provide the following:
- a programming and command interpreter environment
- a language comparable to BASIC or FORTRAN
- a set of system diagnostic and utility commands

With no application to drive, the processor is best suited to compile and execute functions for numerical computations. Programs or macros can be structured by grouping function definitions and command statements together in a file. One function in the group can serve as a main( ) or controlling executive, and a call to that function will initiate a program run.

C2. Program Structure

C2.1 Translation Units

A translation unit defines a set of objects that are translated as a group by the processor. A translation unit is composed of the entire contents of a command stream. The current system defines and supports streams in the form of:

- a single line of input entered at the command line
- multiple line contents of a disk file

Typically, single line streams generated at the command line would be input by the user to issue commands as they interact with the system. Disk files would usually contain function definitions and command statements to structure programs, form libraries, define startup environments, etc.

C2.2 Lifetime

Lifetime defines the duration or existence of an object. Lifetime can be classified as global or local.

- Objects with global lifetime exist for the entire duration of the application.
- Objects with local lifetime exist only during the lifetime of the object they occur in. Systems with dynamic storage allocation allocate and deallocate memory each time a function is called. Parameters, local variables, and local arrays would have local lifetimes.

The system currently does not support any objects with local lifetime. All storage is static in class; once an object has been created, it exists until the main application program is exited.

C2.3 Scope

Scope defines the visibility of an object; the environment space where it can be referenced from. Scope can be classified as global, external, or local.

- Functions and user defined types are global in scope. Objects with global scope can be referenced inside a function definition in compile mode or externally in command mode.
- Variables and arrays declared outside of a function are external in scope. Objects with external scope have no scope within a function. The EXTERN keyword is provided to explicitly declare an object within a function as being external, thus making it accessible to the function.
- Variables and arrays declared within a function definition are local in scope. Objects with local scope have scope only within the function they are declared in.

C2.4 Linkage

Linkage defines the uniqueness of an object with respect to translation units. Linkage can be classified as external, internal, and none.

- Objects with external or global scope have external linkage with respect to translation units. References to an object with external linkage from different translation units refers to the same object.
- Objects with internal linkage are unique to the translation unit they were created in. References to an object with internal linkage from different translation units refer to different objects. The system currently does not support internal linkage.
- Objects with local scope have no linkage. References to an object with no linkage can occur only in a function block and refer to the object declared in that function block.

C3. Language Elements

C3.1 Tokens

Tokens are the smallest language elements discernible to the system. Tokens are identifiers, numbers, operators, special characters, etc. The lexical analysis phase of the compiler has the job of constructing valid language tokens from the source text stream.

C3.2 Comments

Single and multiple line comments are supported using standard C/C++ conventions. Single line comments start with a "//" and continue to the end of the line. Multiple line comments start with "/*" and end with "*/". Comments are ignored by the compiler and stripped from the source text stream during lexical analysis.

Example: FLOAT x,y,z // This is a single line comment

```
/* This is a multiple
line comment */
```

C3.3 Keywords

The language has a set of reserved keywords that cannot be used as names for identifiers. These words are case sensitive so, for example, integer would be a valid identifier name, INTEGER would be illegal since it is a reserved word. Listed below are the reserved keywords. Lowercase keywords are diagnostic and utility commands used for development.

ABS	**ACOS**	**AND**	**ASIN**	**ATAN**	**BREAK**	**CASE**
CLOSE	**COS**	**COSH**	**DEFAULT**	**DEFINE**	**DELETE**	**ELSE**
ELSEIF	**END_DEFINE**	**ENDIF**	**ENDSWITCH**	**ENDWHILE**	**EXP**	**EXTERN**
FLOAT	**FOR**	**FTOA**	**GLOBAL**	**IF**	**INPUT**	**INTEGER**
LET	**LN**	**LOAD**	**LOCAL**	**LOG**	**NEXT**	**NOT**
OPEN	**OR**	**PRINT**	**REPEAT**	**RETURN**	**SIN**	**SINH**
SQRT	**STRING**	**SWITCH**	**SYMBOL**	**SYSTEM**	**TAN**	**TANH**
TRANSLATE	**TYPEDEF**	**UNTIL**	**WHILE**	**ansi**	**clock**	**destruct**
dump	**dumpt**	**exit**	mem	**outbuf**	**scroll**	**til**

C3.4 Identifiers

Identifiers are names, or lexemes, given to variables, arrays, functions, or user defined types. A valid name can be any combination of letters, digits, and underscores, but must begin with a letter or underscore and cannot duplicate a system keyword or previously defined identifier. Identifiers are case sensitive. There is no restriction on name length.

Example:

```
FLOAT force1            // FLOAT variable declaration with identifier name force1
INTEGER element_no      // INTEGER variable declaration with identifier name element_no

DEFINE FLOAT sqr(FLOAT x)  // function definition with identifier name sqr and parameter
    RETURN x^2             // identifier name x
END_DEFINE

TYPEDEF POINT {FLOAT x, FLOAT y}    // user defined type with identifier name POINT
                                    // and member identifier names x and y
```

C3.5 Constants

A constant is a number or string literal. String literals must be enclosed within quotes and cannot contain a quote as part of the string. Spaces are treated as characters in a string literal.

Example:

```
FLOAT x,y,z
INTEGER i
STRING s1
LET x=3.14159          // assignment of floating point number 3.14159 to FLOAT variable x
LET y=0.455E-2         // assignment of floating point number 0.455E-2 to FLOAT variable y
LET z=1.5e3            // assignment of floating point number 1.5e3 to FLOAT variable z
LET i=1234             // assignment of integer number 1234 to INTEGER variable i
LET s1="Load case 1"   // assignment of string literal "Load case 1" to STRING variable s1
LET x = 3.5*SIN(0.7)   // floating point number constants 3.5 and 0.7 used in an expression
```

C3.6 Operators

Operators are special tokens in the language used to perform mathematical or special operations on values, identifiers, or expressions. Table C.1 lists the system operators and associativity. Precedence relates directly to row position in the table. The first row has the highest precedence; the last row has the lowest. Operators in the same row have the same precedence.

Operator	Operation	Associativity
( ) [ ] { } : .	Expression	L>R
+ - NOT	Unary	L>R
^	Exponentiation	R>L
* /	Multiplicative	L>R
+ -	Additive	L>R
< <= > >=	Relational	L>R
= <>	Equality	L>R
AND	Logical	L>R
OR	Logical	L>R
=	Assignment	R>L
,	Sequencing	L>R

Table C.1: Operators and precedence.

C4. Types And Declarations

C4.1 Base Types

Three base data types are provided in the language:

- **FLOAT** double precision floating point number, e.g. 3.14159, -2.5E-2.0
- **INTEGER** integer number, e.g. 1024, -32
- **STRING** character string, e.g., "Hello world"

FLOAT and **INTEGER** data types are direct equivalents of the 'C' language data types **double** and **int**. Range limits are compiler dependent but guaranteed to be at least [1E-37,1E37] for double and [-32767,32767] for int.

STRING data type has no direct 'C' language equivalent. There is no limit on character string length. However, since the maximum token size allowed by the system is 256 characters, string literals are restricted to this length. Concatenation of string variables is a way around this limit using the overloaded operator "+".

C4.2 User Defined Types

The TYPEDEF statement allows the creation of user defined types similar to *structures* in 'C' or *records* in other languages. A user defined type forms a structure used to allocate, division, and access a storage area where one or more pieces of data are stored as a set. Each set member is given a type and name. An object declared with a user defined type is actually a set. Access is accomplished using the "dot" operator as in 'C'.

For example, suppose a variable r is declared as type POINT; **POINT r**. If type POINT contains as members two floating point data types with names x and y, then access to either one is accomplished by **r.x** or **r.y**.

Once defined, variables and arrays can be declared using the new type just as if it were a base type provided in the language. A user defined type may contain members of previously defined user defined types. The assignment operator is overloaded to allow simple assignment of variables or array elements of the same user defined type. All other operators must be applied to a member that is a base type. Functions may return a user defined type for a return value; and in addition, a function may be called using the dot operator to access a member of the function's return value.

Syntax: **TYPEDEF** identifier **{** ***type*** identifier [{, ***type*** identifier}] **}**

Example:

```
TYPEDEF POINT { FLOAT x, FLOAT y}     // defining the type POINT with members
                                      // FLOAT x and FLOAT y
TYPEDEF LINE { POINT a, POINT b }     // defining the type LINE with members
                                      // POINT a and POINT b
FLOAT s,t                             // declaration of FLOAT variables s and t
POINT p1,p2,p3[5]                     // declaration of POINT variables p1,p2, array p3
LINE L1                               // declaration of LINE variable L1
LET p1.x=1.0, p1.y=2.5                // assignment of values to p1 and p2
LET p2.x=5.25, p2.y=11.3              // members
LET L1.a=p1, L1.b=p2                  // assignment of values to L1 members
LET s=p1.y, t=L1.a.y                  // assignment of values to s and t
LET p3[5]=p1                          // assignment of values to p3[5] members
LET p3[5]=p1*2.0      // ERROR: attempt to apply multiplication operator to a user
                      // defined type
```

```
LET p3[5].x=p1.x*2.0    // OK: apply multiplication operator to members that are base
LET p3[5].y=p1.y*2.0    // types
LET p1=func()           // assignment of function that returns a POINT value to POINT
                        // variable p1
LET s=func().x          // assignment of function return value member FLOAT x to
                        // FLOAT variable s
```

C4.3 Declarations

The declaration statement reserves storage for a new variable or array and creates a symbol table entry for it. All variables and arrays must be *declared* before using them. Multiple declarations of the same type are allowed following the type specifier provided the identifiers are separated by commas. A variable or array cannot be initialized in its declaration.

Function declarations are part of the function definition; no separate declaration is required as is the case with variables and arrays.

Syntax: ***type*** identifier [{,identifier}]

Example:

```
FLOAT x,y[3],z[2,4]     // FLOAT variable and array declarations
INTEGER i,j,k           // INTEGER variable declarations
STRING s1,s2,s3         // STRING variable declarations
TYPEDEF POINT {FLOAT x, FLOAT y}
POINT p1[3],p2[4]       // POINT array declarations

FLOAT mag = 3.25        // ERROR: Attempt to initialize a variable during declaration

FLOAT mag               // OK: declaration first, then assignment to initialize the variable
LET mag = 3.25
```

C4.4 Initialization

Any object that requires storage is initialized when the storage is created. This includes local and external variables and arrays, function parameters and return values.

- FLOAT storage initialized to nans
- INTEGER storage initialized to 0
- STRING storage initialized to null

C4.5 Storage

All storage is static in class and exists for the duration of the application. Function parameters, local variables, and local arrays within functions will retain their storage and values after the function is exited.

C5. Variables And Arrays

A variable or array is a reference to an area in memory where data is stored and retrieved. Associated with each variable or array is an address and value. Address refers to the starting point in memory where the data is stored and value refers to what is actually stored in memory.

Base type variables reference a single piece of data. Base type arrays reference a set of data and access set members using the array operator "[ ]".

User defined type variables reference a set of data and access set members using the dot operator ".". User defined type arrays reference sets of data and access sets using the array operator "[ ]" and set members using the dot operator ".".

Variables and arrays must be declared before using them in an expression or attempting to assign a value to them. A valid name can be any combination of letters, digits, and underscores, but must begin with a letter or underscore and cannot duplicate a system keyword or previously declared identifier. There is no restriction on name length.

C5.1 Variables

Syntax: identifier

Example:

```
TYPEDEF POINT { FLOAT x, FLOAT y}     // defining the type POINT with members
                                      // FLOAT x and y
FLOAT xcoor, ycoor      // declaration of FLOAT variables xcoor and ycoor
POINT vector            // declaration of POINT variable vector
INTEGER i1,i2           // declaration of INTEGER variables i1 and i2
STRING _my_string       // declaration of STRING variable _my_string
LET xcoor = 2.5         // assignment of floating point number constant 2.5 to FLOAT
                        // variable xcoor
LET ycoor = xcoor^2     // assignment of floating point expression result to FLOAT variable
                        // ycoor
LET vector.x = xcoor    // assignment of floating point values to POINT variable vector's
                        // members
LET vector.y = ycoor    // FLOAT x and FLOAT y
```

C5.2 Arrays

Single and multidimensional arrays are supported up to a maximum of 10 dimensions. By default, lower array indices implicitly start at 1. However, explicit declaration of the lower indice using the colon operator ":" will override the default. Array indices must be specified using integer constants during declaration. Once declared, any expression returning an integer can be used to specify an indice.

- Initialization of array elements with a user specified value is currently not supported in the language.
- Runtime indice calculation and array bounds checking are supported by the compiler.

Syntax: identifier[[integer:]integer[{,[integer:]integer}]] // declaration syntax

Syntax: identifier[integer expression[{,integer expression}]] // assignment or expression syntax form
identifier // omission of the array operator [ ]
// causes the compiler to treat an array
// as a variable and access is restricted
// to the first element of the array

Example:

```
FLOAT x[3],y[3]          // declaration of 1x3 FLOAT arrays x and y with indice range 1-3
FLOAT point[3,2]         // declaration of 3x2 FLOAT array point with indice ranges 1-3,1-2
FLOAT temp[-5:5,0:10] // declaration of 11x11 FLOAT array temp with indice ranges
                         //  -5-5, 0-10
STRING tags[0:5]         // declaration of 1x6 STRING array tags with indice range 0-5
INTEGER i,j
LET i=-3, j=7
LET x[2] = 5.25
LET temp[i,j] = x[2] // assignment of FLOAT value 5.25 to FLOAT array temp[-3,7]
LET x = x[2]         // assignment of contents of x[2] to x[1] omitting the array operator [ ]
```

C6. Assignments, Operators, and Expressions

C6.1 Assignments

The assignment statement *assigns* the resulting value of an expression to a variable or array element. The keyword **LET** is optional. Multiple assignment statements are allowed on the same line separated by commas. However, due to the free format of the language, omission of the comma separators will still produce the desired result.

Multiple assignment (not multiple assignment statements) is currently not supported by the compiler; For example, LET x=y=z=3.0. The compiler will evaluate this expression, however, the first "=" is interpreted as the assignment operator, the remaining are interpreted as equality operators; e.g., x = ((y=z)=3.0) resulting in x=0.

Syntax: [LET] identifier=expression [{,identifier=expression}]

Example:
```
x=5.0, y[2]=x^2, z[1,1]=(x^2+y[2]^2)^0.5
LET s1="Hello ", s2="world", s3=s1+s2
LET p1[2].x=2.0*pi
x=sqrt(45.0)
```

C6.2 Operators

The operators supported by the language are shown in Table C.2.

Arithmetic	Relational	Equality	Logical
+ addition/positive	< less than	= equal to	NOT logical NOT
- subtraction/negative	<= less than or equal to	<> not equal to	AND logical AND
/ division	> greater than		OR logical OR
* multiplication	>= greater than or equal to		
^ exponentiation			

Expression	String	Sequencing
( ) expression/function	+ concatenation	, sequencing
[ ] array		
{ } type member set		
. member access		
: lower array indice		
= assignment		

Table C.2: Operators.

C6.3 Expressions

Expressions formulate a sequence of operators and operands that when evaluated, return a result. Although expressions are normally associated with assignment statements, they occur throughout the language in many places:

- assignment statements
- array indice specification
- function parameters
- conditional statements
- iterative statements
- function return values

Expression operands include constants, variables, arrays, system and user defined functions. Operators include arithmetic, relational, equality, logical, expression, and string.

```
Example: LET x = COS(2*w) - LN(x)   // assignment expression
         LET y = cam[i+j]           // assignment expression with an array indice integer expression
         curve(x,SIN(x))            // function call with parameter expressions
         IF(x<y)                    // boolean expression in the conditional IF statement
             :
         ENDIF
         WHILE(x<y)                 // boolean expression in the iterative WHILE statement
             :
         ENDWHILE
         DEFINE FLOAT sqr(FLOAT x)
             RETURN x^2             // floating point expression used to return a function value
         END_DEFINE
```

C6.4 Type Conversion

Type conversion is implicitly supported by the compiler to cast integer values to floating point values during assignments and expression evaluation. Assignment conversion occurs when an integer value is assigned to a floating point variable, array, or function parameter. Expression conversion occurs during the evaluation of expressions involving integers and floats. Binary operations involving one float and one integer will cast the integer to a float and then perform the operation. Integer division and exponentiation are undefined in the current system so integers used in these operations are cast to floats.

- Implicit casting requires an additional operation and should be avoided in compiled functions where speed is critical.
- There is no implicit conversion of floats to integers and currently no explicit casting function is provided in the system to do this.

```
Example: FLOAT x,y
         INTEGER i
         LET x = 5      // INTEGER constant 5 is implicitly cast to a FLOAT and assigned to FLOAT
                        // variable x
         LET y = 5.0    // FLOAT constant 5.0 is assigned to FLOAT variable y, no cast required
         LET i = 2      // INTEGER constant 2 is assigned to INTEGER variable i, no cast required
         LET x = i      // value of INTEGER variable i is implicitly cast to FLOAT and assigned to
                        // FLOAT variable x
         LET i = x      // ERROR: attempt to assign a FLOAT value to an INTEGER variable
         LET x = i/2    // division operator causes cast of value of INTEGER variable i and
                        // INTEGER constant 2 to FLOATS before division operation is performed
```

C7. Program Control Flow Statements

Structured programming is fully supported in the language using conditional and iterative control statements (looping and branching). The system supports the following control statements:

Conditional:	**Iterative:**
IF Statement	FOR Statement
SWITCH Statement	WHILE Statement
	REPEAT Statement

- Control statements can be nested to any level.
- Control statements are valid in compile mode only.
- The BREAK statement is supported to allow an immediate exit from an iterative or SWITCH block.

C7.1 Conditional Statements

Conditional or branching statements alter program flow based on whether a condition or set of conditions is true or false, or equivalent.

C7.1.1 IF Statement

The IF statement forms a structure where sets of statements can be grouped and a means to branch to a particular set based on a conditional test.

Syntax: IF(expression) statement_list [{ELSEIF(expression) statement_list}] [ELSE statement_list] ENDIF

In its simplest form, the IF...ENDIF statement evaluates an expression and executes the enclosed statement list if the expression is true (nonzero). If the expression is false (zero), program control transfers to the first statement following the ENDIF keyword.

Example:
```
angle=ATAN(y/x)
IF(x<0 AND y>0)
  angle=angle+pi
  quadrant=2
ENDIF
```

More than one conditional test can be made using the form IF...ELSEIF...ENDIF. If the IF expression is false, program control transfers to the first ELSEIF and evaluates its expression. If the ELSEIF expression is false, program control transfers to the next ELSEIF and evaluates its expression, and so on. A sequential evaluation of expressions is performed until an expression yields true or the block is terminated by the ENDIF. Once an expression yields true, the enclosed statement list is executed, then program control transfers to the first statement following the ENDIF keyword.

Example:
```
angle=ATAN(y/x)
IF(x<0 AND y>0)
  angle=angle+pi
  quadrant=2
ELSEIF(x<0 AND y<0)
  angle=angle+pi
  quadrant=3
ELSEIF(x>0 AND y<0)
  quadrant=4
ENDIF
```

The ELSE keyword provides unconditional execution of a set of statements to occur if no preceding IF or ELSEIF expression yielded true. Forms are IF...ELSE...ENDIF and IF...ELSEIF...ELSE...ENDIF.

Example:
```
angle=ATAN(y/x)
IF(x<0 AND y>0)
 angle=angle+pi
 quadrant=2
ELSEIF(x<0 AND y<0)
 angle=angle+pi
 quadrant=3
ELSEIF(x>0 AND y<0)
 quadrant=4
ELSE
 quadrant=1
ENDIF
```

C7.1.2 SWITCH Statement

The SWITCH statement forms a structure where sets of statements can be grouped and a means to branch to a particular set based on an equivalence test. A SWITCH statement evaluates a single SWITCH expression, then sequentially evaluates a list of CASE expressions until the result of a CASE expression exactly matches the result of the SWITCH expression. Program control then transfers to the first statement of the statement set enclosed by the matching CASE. Once a CASE expression match is found, no further CASE expressions are evaluated. If no CASE match is found, program control transfers to the first statement following the ENDSWITCH keyword or, optionally, a DEFAULT case can be specified that will be unconditionally executed if no CASE matches occur.

The BREAK statement is used to cause an immediate exit from a SWITCH statement and transfer program control to the first statement following the ENDSWITCH keyword. The BREAK statement is normally included as the last statement in a CASE statement set. If omitted, program control will transfer to the first statement of the next CASE statement set which may or may not be desirable. Omission of statement sets is commonly used to group multiple CASE's together which share a common statement set.

CASE expressions are not limited to constants as is the case in the 'C' language.

Syntax: **SWITCH**(expression) **CASE**(expression) statement_list [**BREAK**] [**CASE**(expression) statement_list [**BREAK**]] [**DEFAULT** statement_list [**BREAK**]] **ENDSWITCH**

Example:

```
DEFINE INTEGER days(INTEGER month)
  INTEGER numdays
  SWITCH(month)
    CASE(1)          // "fall through" of CASE statements to reach a common statement set
    CASE(3)
    CASE(5)
    CASE(7)
    CASE(8)
    CASE(10)
    CASE(12)
      numdays = 31
      BREAK                  // using BREAK command to exit SWITCH statement
    CASE(4)
    CASE(6)
    CASE(9)
    CASE(11)
      numdays = 30
      RETURN numdays         // using RETURN statement to exit function from within SWITCH
    CASE(2)                  // statement
      RETURN 28              //
    DEFAULT
      numdays = 0
      BREAK                  // BREAK is redundant for DEFAULT case, can be omitted
  ENDSWITCH
  RETURN numdays
END_DEFINE
```

C7.2 Iterative Statements

Iterative or looping statements allow repeated execution of statement sets based on whether a condition or set of conditions is true or false.

C7.2.1 FOR Statement

The FOR statement forms a structure where a set of statements can be grouped and repeatedly executed based on a conditional test. Normally, the FOR statement is used to execute a loop a specified number of times. Loop variable initialization and incrementing are supported by the structure to accomplish this.

Syntax: **FOR**(assignment_list ; expression ; assignment_list) statement_list **NEXT**

where assignment_list is of the form:

identifier=expression [{,identifier=expression}]

The first parameter of the FOR statement is an assignment list that's executed only once prior to evaluating the loop. This assignment list is usually where loop variables are initialized, but other assignments are also permissible.

The second parameter of the FOR statement is the conditional expression. This expression is evaluated and if true, the enclosed statement list is executed. If the expression is false, program control transfers to the first statement following the NEXT keyword. The conditional expression is evaluated at the start of each iteration of the loop.

The third parameter of the FOR statement is an assignment list that's executed after each iteration of the loop. This assignment list is usually where loop variables are incremented, but other assignments are permissible. After execution of the assignment list, program control transfers back to the conditional test to start the next iteration.

The BREAK statement can be used to cause an immediate exit from a FOR statement and transfer program control to the first statement following the NEXT keyword.

Example:
```
DEFINE FLOAT test(INTEGER max, INTEGER iinc, INTEGER jinc)
  INTEGER i,j
  FLOAT x

  FOR(x=0,i=1,j=1;i+j<=max;i=i+iinc,j=j+jinc)
    LET x=x+1
  NEXT

  RETURN x
END_DEFINE
```

C7.2.2 WHILE Statement

The WHILE statement forms a structure where a set of statements can be grouped and repeatedly executed based on a conditional test. The WHILE statement evaluates an expression and executes the enclosed statement list if the expression is true. After execution of the statement set, program control transfers back to the conditional test to start the next iteration. If the expression is false, program control transfers to the first statement following the ENDWHILE keyword.

The BREAK statement can be used to cause an immediate exit from a WHILE statement and transfer program control to the first statement following the ENDWHILE keyword.

Syntax: **WHILE**(expression) statement_list **ENDWHILE**

Example:
```
DEFINE fillarray(FLOAT array[], INTEGER n, INTEGER m, FLOAT t)
  INTEGER i,j
  LET i = 1
  WHILE(i<=n)
    LET j=1
    WHILE(j<=m)
      LET array[i,j] = t
      LET j = j+1
    ENDWHILE
    LET i = i+1
  ENDWHILE
END_DEFINE
```

C7.2.3 REPEAT Statement

The REPEAT statement forms a structure where a set of statements can be grouped and repeatedly executed based on a conditional test. The REPEAT statement is similar to the WHILE statement except that the conditional expression is evaluated after the enclosed statement list is executed. The REPEAT statement first executes the enclosed statement list and then evaluates the conditional expression. If the expression is false, program control transfers back to the first statement of the statement list to start the next iteration. If the expression is true, program control transfers to the first statement following the UNTIL keyword.

The BREAK statement can be used to cause an immediate exit from a REPEAT statement and transfer program control to the first statement following the UNTIL keyword.

Syntax: **REPEAT** statement_list **UNTIL**(expression)

Example:
```
DEFINE fillarray(FLOAT array[], INTEGER n, INTEGER m, FLOAT t)
  INTEGER i,j
  LET i = 1
  WHILE(i<=n)
    LET j=1
    REPEAT
      LET array[i,j] = t
      LET j = j+1
    UNTIL(j>m)
    LET i = i+1
  ENDWHILE
END_DEFINE
```

C7.3 BREAK Statement

The BREAK statement is used to cause an immediate exit from an iterative or SWITCH block and transfer program control to the first statement following the block terminator (NEXT, UNTIL, etc.). In nested blocks, the BREAK statement exits only the block it appears in.

Syntax: **BREAK**

Example:

```
DEFINE fillarray(FLOAT array[], INTEGER n, INTEGER m, FLOAT t)
  INTEGER i,j
  LET i = 1
  WHILE(1)
    LET j=1
    REPEAT
      LET array[i,j] = t
      LET j = j+1
    UNTIL(j>m)
    LET i = i+1
    IF(i>n)
      BREAK;        // Break causes an immediate exit from WHILE loop
    ENDIF
  ENDWHILE
END_DEFINE
```

C8. Functions

A function or macro forms a structure where statements can be grouped together and executed by referencing the function by name. Functions can access most system commands, call other functions, be used in expressions to return a value, and be called as a command to execute some task.

- Functions can return a single value of any type; base or user defined.
- Parameters can be passed as "call by value" or "call by reference".
- Variables and arrays declared in functions are by default local to the function.
- The EXTERN keyword is available to access an external variable or array.
- The RETURN statement is supported to allow an immediate exit from a function and transfer program control back to the first statement following the statement that called the function.
- Recursion is not currently supported, so a function cannot call itself within the body of the function. You can, however, call a function and use that same function in its parameter list; e.g., SQRT(SQRT(16)).
- Functions cannot be defined within functions.

Functions are compiled into fully analyzed threaded code so no extensive interpretation is required at runtime. Test code has shown execution speeds roughly 3 times faster than equivalent code run in Microsoft QuickBasic.

C8.1 Definition

A function definition declares and defines a function.

A function definition includes:
- the DEFINE keyword
- an optional return type specifier if the function is to return a value
- a function identifier name
- an optional parameter list
- the statement list or body of the function
- the END_DEFINE keyword

Syntax: DEFINE [*type*] identifier[([*type* identifier [{ ,*type* identifier }]])] statement_list
END_DEFINE

Example:
```
TYPEDEF POINT {FLOAT mag, FLOAT ang}

DEFINE POINT polar(FLOAT x, FLOAT y)  // function definition with identifier name polar,
  POINT r1                            // return type specifier POINT, and parameters
  r1.mag=SQRT(x^2+y^2)                // FLOAT x and FLOAT y
  r1.ang=ATAN2(x,y)
  RETURN r1
END_DEFINE

DEFINE FLOAT pi()          // function that takes no parameters but returns a value
  RETURN 3.14159
END_DEFINE

DEFINE FLOAT pi            // same function but with parenthesis omitted; no parameters
  RETURN 3.14159
END_DEFINE
```

```
DEFINE pause                // function that takes no parameters and returns no value
  INTEGER i
  FOR(i=0;i<10000;i=i+1)
  NEXT
END_DEFINE
```

C8.2 Parameters, Local Variables and Arrays

Function parameters are declared in the function definition parameter list.

- Each parameter must have a type and unique identifier
- A parameter identifier is local to the function
- A variable or array may be passed as a parameter to a function
- Arrays are always passed as "call by reference"
- Variables are passed either as "call by reference" or "call by value"

Call by value passes the function a copy of a value and has no effect outside of the function. Call by reference passes the address of an external variable or array to a function and the function operates on the external variable or array.

To declare a parameter variable as "call by reference", prefix the identifier with "&" in the parameter list. Since arrays are always passed as "call by reference", the "&" is optional. Array parameters can be declared with or without the "[ ]" operator in the parameter list.

Example:
```
// Function definition with two array parameters and one reference variable. Note optional
// omission of "[ ]" operator for array parameter a2.

DEFINE FLOAT test(point a1[ ], INTEGER &m, STRING a2)
:
:
END_DEFINE
```

Function variables and arrays are declared in the body of the function.

- Variables and arrays are by default local to the function
- The EXTERN keyword is used to declare a function variable or array as external
- Variable and array identifiers must be unique within the function and cannot duplicate the function's parameter identifiers, a system keyword, or a base or user defined type.
- A variable or array identifier may duplicate a function name.

To declare a variable or array in a function as external, prefix the declaration of the identifier with the keyword EXTERN. For example, EXTERN FLOAT x declares x to be an external floating point variable. EXTERN STRING list[ ] declares list to be and external array of string variables.

Example:
```
TYPEDEF point { FLOAT x, FLOAT y }
point x[2,3], y[3]
STRING s[4,2]
INTEGER j

DEFINE FLOAT pi()
  RETURN 3.14159265358979323846
END_DEFINE
```

```
DEFINE FLOAT sqr(FLOAT x)
  RETURN x^2.0
END_DEFINE

// Function with reference and local parameters, external and local declarations
// Also case of a local variable overriding a function name

DEFINE FLOAT test(point a1[], INTEGER &m, STRING a2[])
  EXTERN point y[]          // declaration of external point array y[] using EXTERN keyword
  FLOAT cam[1,2,3]          // declaration of local FLOAT array cam[1,2,3]
  INTEGER i,j,k             // declaration of local INTEGER variables i, j, and k
  LET i=1,j=2,k=3       // assignment of integer constants to local INTEGER variables i,j,k

  LET y[i].x=1.1, y[i].y=1.2      // assignment or values to external array y[]
  LET y[j].x=2.1, y[j].y=2.2
  LET y[i+j].x=3.1, y[i+j].y=3.2

  LET a1[2,2].x=2.21              // assignment of value to external array x[2,2]
  LET a2[2,2]="Test string"       // assignment of value to external array s[2,2]
  LET m=37                        // assignment of value to external variable j

  FLOAT pi LET pi=5       // Declaring a local variable "pi" with same name as
                          // function pi(). Local variable has scope from this point
                          // on until end of function block.
  RETURN sqr(pi)          // function returns value of 25.0, not 9.869604401
END_DEFINE

test(x,j,s)               // call function passing parameters x,j,s
```

C8.3 RETURN Statement

The RETURN statement is used to cause an immediate exit from a function and transfer program control back to the point where the function was called from. An optional return expression will be evaluated and the result returned from the function call.

- Any number of RETURN statements may appear in a function body
- A RETURN statement is optional for a function that returns no value and cannot include a return expression
- A RETURN statement is mandatory for a function that returns a value and must include a return expression

Syntax: **RETURN** [expression]

Example:

```
DEFINE FLOAT abs(FLOAT x)     // function to calculate absolute value
  IF(x<=0)
    RETURN -x                 // x is negative; return -x
  ENDIF
  RETURN x                    // x is positive; return x
END_DEFINE
```

C8.4 Invocation

Functions can be used in expressions and are treated just as if they were a variable being used in the expression. A function is executed and its return value is used in the expression for calculation as well as the return value type for type checking and implicit type conversion. Functions that have no return value are implicitly typed VOID and will generate a type mismatch error if used in an expression. Functions returning a user defined type can be invoked using the dot operator to access a member of the return value.

Functions do not have to be used in expressions and can be called as stand alone commands. Any return value will be ignored by the system in this case.

Syntax: identifier[([expression [{ ,expression }]])]

Example:

```
TYPEDEF POINT {FLOAT mag, FLOAT ang}
POINT pcoor
FLOAT x1,x2,magnitude
x1 = 1.5
x2 = 3.25
pcoor = polar(x1,x2)            // call to function to calculate polar coordinates
magnitude = polar(x1,x2).mag    // call to function using dot operator to access mag value
pause                           // calling a function as a command outside of an expression
```

C9. Symbolic Constants

Symbolic constants are user defined tokens that have a replacement string associated with them. During translation, any occurrence of a symbolic constant results in a direct substitution of the replacement string defined for that constant.

Syntax: **SYMBOL** identifier string_literal

Example:
```
SYMBOL pi "3.14159"   // defines a symbolic constant named pi with replacement string
                      // "3.14159"
x = 2 * pi            // symbolic constant used in an expression
```

C10. Miscellaneous Commands

C10.1.1 SYSTEM Statement

The SYSTEM command allows the user to shell out of the translator and execute another program.

Syntax: **SYS**TEM string_expression

Example: SYSTEM "dir" // shells out to the DOS command line and executes a directory listing

C10.1.2 DELETE Statement

The DELETE command allows the user to delete variables, arrays, functions, symbolic constants, and user defined types.

Syntax: **DEL**ETE string_expression_list

Example: DELETE "x","y" // delete the objects x and y

C10.1.3 LOCAL Statement

The LOCAL command sets the scope of the symbol table search routines local to a function with respect to variables and arrays. This command is used in conjunction with the TRANSLATE command as a temporary solution to the absence of pointers in the current language.

Syntax: **LOCAL** string_expression

Example: LOCAL "myfunc" // sets symbol table routines local to function "myfunc"

C10.1.4 GLOBAL Statement

The GLOBAL command restores the scope of the symbol table search routines with respect to variables and arrays. This command is used in conjunction with the TRANSLATE command as a temporary solution to the absence of pointers in the current language.

Syntax: **GLOBAL**

C10.1.5 TRANSLATE Statement

The TRANSLATE command exploits the reentrant capabilities of the translator. At runtime, a call to TRANSLATE(string_expression) will suspend the current operation of the translator and then reenter the translator to process the contents of the string_expression. Once this operation is complete, control returns to continue execution of the program that issued the TRANSLATE call.

Syntax: **TRANSLATE**(string_expression)

Example:
```
STRING s
s = "DEFINE sqr(FLOAT x) RETURN x^2 END_DEFINE"    // string variable containing a
                                                   // function definition
PRINT sqr(4)      // ERROR! No function sqr() exists at this point
TRANSLATE(s)      // process string s to compile new function sqr() into system
PRINT sqr(4)      // call new fuction sqr() to calculate and print 4^2
```

C11. Specialized Command Sets

C11.1 Mathematical

The language supports a core set of mathematical functions summarized below:

SIN(x)	sine of x		
COS(x)	cosine of x		
TAN(x)	tangent of x		
ASIN(x)	inverse sine of x	$-1 \le x \le 1$	$-\pi/2 \le f(x) \le \pi/2$
ACOS(x)	inverse cosine of x	$-1 \le x \le 1$	$0 \le f(x) \le \pi$
ATAN(x)	inverse tangent of x		$-\pi \le f(x) \le \pi$
SINH(x)	hyperbolic sine of x		
COSH(x)	hyperbolic cosine of x		
TANH(x)	hyperbolic tangent of x		
ABS(x)	absolute value $\|x\|$		
SQRT(x)	square root of x	$x \ge 0$	$f(x) \ge 0$
LOG(x)	base 10 logarithm of x	$x > 0$	
LN(x)	natural logarithm of x	$x > 0$	
EXP(x)	exponential function e^x		

x used above represents any expression that returns a floating point value.

All the mathematical functions return a floating point value and can be used in any expression just the same as a user defined function.

Syntax: identifier(expression)

Example:
```
FLOAT y,Xo,s,w,t,phi
   :
LET y = Xo*EXP(-s*w*t)*SIN(SQRT(1-s^2)*w*t+phi) // expression using system functions
                                                // EXP(), SIN(), and SQRT()
FLOAT x,y
LET y = myfunc(SIN(x)+x^2)    // function call using the system function SIN() in a
                              // parameter expression
```

C11.2 Utility

Utility functions are summarized below:

FTOA(x) convert floating point number x to a string and return the string

C11.3 I/O

C11.3.1 LOAD Statement

The LOAD command reads a file from disk into the system command stream being processed by the compiler. It's normally invoked from the command line; however, it can be included in a macro file to facilitate chaining of files during loading. The LOAD command cannot be used in a function.

Syntax: **LOAD** "[drive][path]filename"

Example:
```
LOAD "C:\macros\custom.mac"   // loads and processes the macro file custom.mac
                              // located in the macros directory on drive C:
LOAD "tools.mac"              // loads and processes the macro file tools.mac located
                              // in the  current working directory
```

C11.3.2 OPEN Statement

The OPEN statement is used to open a disk file for input or output.

Syntax: **OPEN** #channel, mode, filename

channel	Integer expression in range 1 to 255.
mode	String expression specifying read, write, or append mode (r, w, or a.)
filename	String expression specifying filename and optional drive/path.

Example:
```
OPEN #1,"w","test1.txt"      // opens file test1.txt in write mode and assigns file to channel 1

INTEGER channel
STRING mode, file
channel =10
mode = "r"
file = "c:\user\test2.txt"
OPEN #channel,mode,file      // opens file c:\user\test2.txt in read mode and assigns file to
                             // channel 10
```

C11.3.3 CLOSE Statement

The CLOSE statement is used to close files that were previously opened with the OPEN command.

Syntax: **CLOSE** #channel
CLOSE

channel Integer expression in range 1 to 255.

Example:
```
CLOSE #3      // closes the file assigned to channel 3

CLOSE         // closes all open files
```

C11.3.4 INPUT Statement

The INPUT statement reads input from the keyboard or disk. Input values must be separated with commas.

Syntax: **INPUT** prompt, variable_list
INPUT #channel,variable_list

prompt	String expression used to display a user-defined prompt to the screen.
channel	Integer expression in range 1 to 255.
variable_list	List of variables to assign input values to.

Example:
```
INPUT "Enter x,i,s: ",x,i,s     // displays the prompt "Enter x,i,s: " to the screen and waits for
                                // the user to key in values for x, i, and s

INPUT #1,x1,i1,s1               // reads input from the currently open file assigned to channel 1
                                // and assigns the input values to variables x1, i1, and s1
```

C11.3.5 PRINT Statement

The PRINT statement writes output to the screen or disk. Output expressions must be separated with commas.

Syntax: **PRINT** expression_list
PRINT #channel,expression_list

expression_list	List of expressions to print.
channel	Integer expression in range 1 to 255.

Example:
```
FLOAT x
x = 1.0
PRINT "The sine of ",x," = ",SIN(x)      // prints "The sine of 1 = 0.841471" to the screen
PRINT #2,"The sine of ",x," = ",SIN(x)   // prints "The sine of 1 = 0.841471" to the currently
                                         // open file assigned to channel 2
```

C11.4 Diagnostic

A set of diagnostic and utility tools necessarily evolved during the development of the compiler. Although not intended for inclusion in the final application, the tools are documented here and retained in the current system for use in future development.

dump Toggle display of entire symbol table to the screen on or off.

dump xxxxxx Display sections of the symbol table to the screen. From left to right, x corresponds to variable list, array list, function list, reserved word list, symbolic list, and type list. For example:

```
dump 111111   display entire symbol table
dump 000000   turn off entire symbol table
dump 110001   display variable, array, and type lists
dump 001000   display function list
dump 000001   display type list
```

scroll Toggle scrolling of output on or off

outbuf Toggle display of intermediate language output from parser on or off

dumpt Toggle display of threaded code during function compilation on or off

til Toggle between front and back end compiler environments.

destruct Toggle display of symbol table object destructors on or off.

ansi Toggle ansi display on or off (use if ANSI.SYS driver is present)

mem Displays the TIL dictionary size, bytes used, and bytes free.

clock Toggles a timer on and off. This command is called twice; the first call starts the clock, the second call stops the clock and displays the elapsed time to the nearest 1/100 sec. Useful for timing routines during development optimization. For example:

```
com> clock myfunc() clock <ENTER>
Elapsed Time: 1.65 sec
com>
```

The function myfunc() took 1.65 sec to execute.

exit Exit the command processor

C12. Operations Guide

C12.1 System Requirements

- IBM or 100% compatible computer running MS-DOS 3.3 or above
- 80386 or higher processor with math coprocessor
- One megabyte of RAM or more
- One floppy disk drive; program supplied in 3.5 and 5.25 inch formats

C12.2 System Configuration

Recommend installing the ANSI.SYS driver to enable color output of the Symbol Table listings. Include the following statement in the config.sys file (or similar if [drive:][path] is different):

DEVICE=C:\DOS\ANSI.SYS

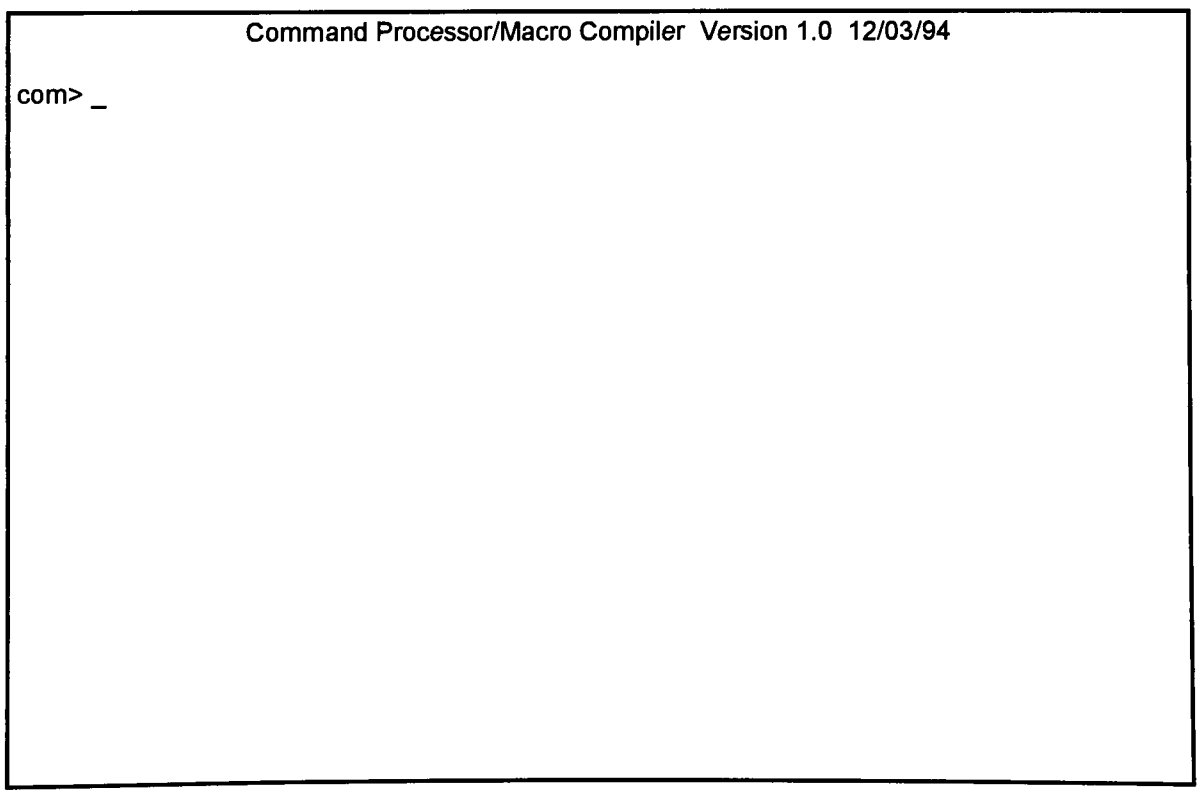
C12.3 Running the Command Processor

From the DOS prompt, type:

[drive:][path]**cp** [[drive:][path]filename]

where *filename* is an optional macro file that will be loaded and processed by the compiler at startup.

If no macro file is loaded, the initial startup screen appears as shown below:



A title banner is displayed showing the current version and release date of the system.
The command line prompt is displayed and the system is ready for user input.

Pressing <ENTER> at this point will clear the startup screen and display a full Symbol Table Listing above the command line as shown below. This listing reflects the starting state of the compiler's symbol table; reserved words and base types, no variables, arrays, functions, symbolic constants, or user defined types.

```
SYMBOL TABLE LISTING:
[Variable List]
[Array List]
[Function List]
[Reserved List]
 LET          TYPEDEF      DEFINE       END_DEFINE   RETURN       EXTERN
 IF           ELSEIF       ELSE         ENDIF        FOR          NEXT
 WHILE        ENDWHILE     REPEAT       UNTIL        SWITCH       CASE
 DEFAULT      ENDSWITCH    BREAK        INPUT        PRINT        OPEN
 CLOSE        LOAD         FTOA         TRANSLATE    DELETE       SYMBOL
 SYSTEM       NOT          AND          OR           SIN          COS
 TAN          ASIN         ACOS         ATAN         SINH         COSH
 TANH         ABS          SQRT         LOG          LN           EXP
 LOCAL        GLOBAL       dump         scroll       outbuf       dumpt
 til          ansi         destruct     clock        mem          exit
[Symbolic List]
[Type List]
 FLOAT        t:337 su:008
 INTEGER      t:336 su:004
 STRING       t:335 su:004
com> _
```

Entering a command string and pressing <ENTER> or loading a macro file would generate a similar listing that included any updates to the symbol table; variables, arrays, functions, symbolic constants, and user defined types.

To enable color highlighting of the Symbol Table Listing (ANSI.SYS driver must be present), type:

com> **ansi** <ENTER>

```
SYMBOL TABLE LISTING:
[Variable List]
[Array List]
[Function List]
[Reserved List]
 LET          TYPEDEF      DEFINE       END_DEFINE   RETURN       EXTERN
 IF           ELSEIF       ELSE         ENDIF        FOR          NEXT
 WHILE        ENDWHILE     REPEAT       UNTIL        SWITCH       CASE
 DEFAULT      ENDSWITCH    BREAK        INPUT        PRINT        OPEN
 CLOSE        LOAD         FTOA         TRANSLATE    DELETE       SYMBOL
 SYSTEM       NOT          AND          OR           SIN          COS
 TAN          ASIN         ACOS         ATAN         SINH         COSH
 TANH         ABS          SQRT         LOG          LN           EXP
 LOCAL        GLOBAL       dump         scroll       outbuf       dumpt
 til          ansi         destruct     clock        mem          exit
[Symbolic List]
[Type List]
 FLOAT        t:337 su:008
 INTEGER      t:336 su:004
 STRING       t:335 su:004
com> _
```

The Symbol Table Listing is updated each time a command stream is processed. For example, suppose the following code is stored in a macro file called *list.mac* in the current working directory:

```
TYPEDEF POLAR {FLOAT mag, FLOAT ang}

DEFINE FLOAT pi()
  RETURN 3.14159265358979323846
END_DEFINE

DEFINE POLAR RecToPolar(FLOAT x, FLOAT y)
  POLAR r1
  r1.mag = SQRT(x^2+y^2)
  r1.ang = ATAN(y/x)
  IF(x<0)
    r1.ang = r1.ang+pi
  ENDIF
  RETURN r1
END_DEFINE

POLAR r
FLOAT x1, x2, vect[3,2]
```

At the command line, enter:

com> **LOAD "list.mac"** <ENTER>

Processing the above macro file generates the Symbol Table Listing shown below. Note that a portion of the listing has scrolled off the screen.

SYMBOL TABLE LISTING:

```
[Variable List]
 x2              t:337 &v:277088 v: <nans>
 x1              t:337 &v:277044 v: <nans>
 r               t:338 &v:276992 v: <nans> <nans>
[Array List]
 vect            t:337 &v:277156 [1:3,1:2]
 [1,1] <nans>
 [1,2] <nans>
 [2,1] <nans>
 [2,2] <nans>
 [3,1] <nans>
 [3,2] <nans>
[Function List]
 RecToPolar      t:338 &v:276808 v: <nans> <nans> (x:337,y:337)
 pi              t:337 &v:276572 v: <nans> ( )
[Reserved List]
 LET             TYPEDEF         DEFINE          END_DEFINE      RETURN          EXTERN
 IF              ELSEIF          ELSE            ENDIF           FOR             NEXT
 WHILE           ENDWHILE        REPEAT          UNTIL           SWITCH          CASE
 DEFAULT         ENDSWITCH       BREAK           INPUT           PRINT           OPEN
 CLOSE           LOAD            FTOA            TRANSLATE       DELETE          SYMBOL
 SYSTEM          NOT             AND             OR              SIN             COS
 TAN             ASIN            ACOS            ATAN            SINH            COSH
 TANH            ABS             SQRT            LOG             LN              EXP
 LOCAL           GLOBAL          dump            scroll          outbuf          dumpt
 til             ansi            destruct        clock           mem             exit
[Symbolic List]
[Type List]
 POLAR           t:338 su:016 {mag:337,ang:337}
 FLOAT           t:337 su:008
 INTEGER         t:336 su:004
 STRING          t:335 su:004
com> _
```

Once a user becomes familiar with the systems reserved words, turning off the Reserved List is desirable to prevent the screen from scrolling as the symbol table grows with entries.

At the command line, enter:

com> **dump 111011** <ENTER>

The Reserved List is turned off and the Symbol Table Listing generated is shown below.

```
SYMBOL TABLE LISTING:
[Variable List]
 x2              t:337 &v:277088 v: <nans>
 x1              t:337 &v:277044 v: <nans>
 r               t:338 &v:276992 v: <nans> <nans>
[Array List]
 vect            t:337 &v:277156 [1:3,1:2]
 [1,1] <nans>
 [1,2] <nans>
 [2,1] <nans>
 [2,2] <nans>
 [3,1] <nans>
 [3,2] <nans>
[Function List]
 RecToPolar      t:338 &v:276808 v: <nans> <nans> (x:337,y:337)
 pi              t:337 &v:276572 v: <nans> ( )
[Symbolic List]
[Type List]
 POLAR           t:338 su:016 {mag:337,ang:337}
 FLOAT           t:337 su:008
 INTEGER         t:336 su:004
 STRING          t:335 su:004
com> _
```

Eventually the symbol table will grow to the point where scrolling is inevitable. Hitting the <PAUSE> key will freeze a listing temporarily and may be useful. To capture a full listing for detailed examination, run the program with the macro file as a parameter and pipe the output to a file. Then exit the program and examine the file.

Example: At the DOS prompt, run the program cp with the macro file list.mac as a parameter and pipe the output to a file named output.txt.

cp list.mac>output.txt

Then type **exit** <ENTER> to exit the program and return to the DOS prompt to examine the file.

Piping output to a text file can also be used to capture intermediate language output from the parser and threaded code during function compilations when using the **outbuf** and **dumpt** diagnostic tools.

The **scroll** command *will not* prevent a large listing from scrolling. This command toggles a clear screen operation on or off before displaying a Symbol Table Listing. If scroll is off, the screen is cleared each and every time the Symbol Table Listing is displayed. If scroll is on, no clear screen operation is performed and the new listing scrolls up under the old.

scroll should be turned on to observe command lines entered and intermediate language output, threaded code generation, and object destructor listings using the outbuf, dumpt, and del diagnostic tools.

At the command line, enter:

com> **scroll** <ENTER>

Scrolling is enabled and the Symbol Table Listing generated is shown below. Note that the listing has scrolled up under the last listing and the previous command line is still visible.

```
STRING          t:335 su:004
com> scroll
SYMBOL TABLE LISTING:
[Variable List]
 x2             t:337 &v:277088 v: <nans>
 x1             t:337 &v:277044 v: <nans>
 r              t:338 &v:276992 v: <nans> <nans>
[Array List]
 vect           t:337 &v:277156 [1:3,1:2]
 [1,1] <nans>
 [1,2] <nans>
 [2,1] <nans>
 [2,2] <nans>
 [3,1] <nans>
 [3,2] <nans>
[Function List]
 RecToPolar     t:338 &v:276808 v: <nans> <nans> (x:337,y:337)
 pi             t:337 &v:276572 v: <nans> ( )
[Symbolic List]
[Type List]
 POLAR          t:338 su:016 {mag:337,ang:337}
 FLOAT          t:337 su:008
 INTEGER        t:336 su:004
 STRING         t:335 su:004
com> _
```

To disable scrolling, reenter:

com> **scroll** <ENTER>

To dynamically observe the contents of variables and arrays, enter the following statements at the command line:

com> **x1 = -1.0** <ENTER>
com> **x2 = -2.0** <ENTER>
com> **r = RecToPolar(x1,x2)** <ENTER>
com> **vect[1,1] = r.mag** <ENTER>
com> **vect[1,2] = r.ang** <ENTER>

The final Symbol Table listing is shown below.

```
SYMBOL TABLE LISTING:
[Variable List]
 x2              t:337 &v:277088 v: <-2>
 x1              t:337 &v:277044 v: <-1>
 r               t:338 &v:276992 v: <2.23607> <4.24874>
[Array List]
 vect            t:337 &v:277156 [1:3,1:2]
 [1,1] <2.23607>
 [1,2] <4.24874>
 [2,1] <nans>
 [2,2] <nans>
 [3,1] <nans>
 [3,2] <nans>
[Function List]
 RecToPolar      t:338 &v:276808 v: <2.23607> <4.24874> (x:337,y:337)
 pi              t:337 &v:276572 v: <3.14159> ( )
[Symbolic List]
[Type List]
 POLAR           t:338 su:016 {mag:337,ang:337}
 FLOAT           t:337 su:008
 INTEGER         t:336 su:004
 STRING          t:335 su:004
com> _
```

The Symbol Table Listing can be set to display a single function's local variables and arrays.

To observe the local variables and arrays in the function RecToPolar, enter:

com> **LOCAL "RecToPolar"** <ENTER>

The Variable and Array Lists are set locally to the function RecToPolar and the Symbol Table Listing generated is shown below.

```
SYMBOL TABLE LISTING:
[RecToPolar local Variable List]
 r1              t:338 &v:276860 v: <2.23607> <4.24874>
 x               t:337 &v:276796 v: <-1>
 y               t:337 &v:276752 v: <-2>
[RecToPolar local Array List]
[Function List]
 RecToPolar      t:338 &v:276808 v: <2.23607> <4.24874> (x:337,y:337)
 pi              t:337 &v:276572 v: <3.14159> ( )
[Symbolic List]
[Type List]
 POLAR           t:338 su:016 {mag:337,ang:337}
 FLOAT           t:337 su:008
 INTEGER         t:336 su:004
 STRING          t:335 su:004
com> _
```

To exit local display mode, enter:

com> **GLOBAL** <ENTER>

C12.4 Interpreting the Symbol Table Listing

The Symbol Table Listing is a tool to view the current state of the system's symbol table. Variables, arrays, functions, reserved words, symbolic constants, and user defined types are all managed by the symbol table routines. The tool displays variable and array lexemes, types, storage addresses and contents. Function lexemes, return types and storage addresses and contents, parameter lexemes and types, local variables and arrays. The systems reserved word list and symbolic constants. Base and user defined types, type codes, storage requirements, member lexemes and types.

[Variable List]
The Variable List is a display of all the variables defined in the system. The listing shows a variable's name, typecode, storage address, and storage contents.

Format: *variable_name* **t:***typecode* **&v:***storage_address* **v: {<***storage_contents***>}**

Example: **x1** t:337 &v:277044 v: **<-1>**

Variable **x1** is type **FLOAT** with starting storage address at **277044** and contents **-1**.

r t:338 &v:276992 v: **<2.23607>** **<4.24874>**

Variable **r** is type **POINT** with starting storage address at **276992** and contents **2.23607**, **4.24874**.

[Array List]
The Array List is a display of all the arrays defined in the system. The listing shows an array's name, typecode, storage address, indice range, and storage contents.

Format: *array_name* **t:***typecode* **&v:***storage_address* **[***indice***:***indice***{,***indice***:***indice***}]**
[i,j...] <storage_contents>
[i,j+1...] <storage_contents>
[i,j+2...] <storage_contents>
:
[m,n...] <storage_contents>

Example: **vect** t:337 &v:277156 [1:3,1:2]
[1,1] **<2.23607>**
[1,2] **<4.24874>**
[2,1] **<nans>**
[2,2] **<nans>**
[3,1] **<nans>**
[3,2] **<nans>**

Array **vect** is type **FLOAT** with starting storage address at **277156**. The array has 2 dimensions with indice ranges 1:3 and 1:2. Array elements [1,1] and [1,2] contain values **2.23607**, **4.24874;** the remaining have never been assigned a value yet and contain **nans**.

[Function List]
The Function List is a display of all the functions defined in the system. The listing shows a function's name, typecode, storage address, return storage contents, and parameter list.

Format: *function_name* **t:***typecode* **&v:***storage_address* **v:** {<*storage_contents*>} ([*parameter_name*:*typecode*{,*parameter_name*:*typecode*}])

Example: **RecToPolar** t:338 &v:276808 v: <**2.23607**> <**4.24874**> (**x**:337,**y**:337)

Function **RecToPolar** is type **POINT** with return value starting storage address at **276808**. The function has been called and the return storage contains values **2.23607**, **4.24874**. The function has two parameters; **FLOAT** variable **x** and **FLOAT** variable **y**.

[Reserved List]
The Reserved List is a display of all the reserved keywords in the system except base types. Base types are listed separately in the Type List.

Format: *reserved_word*

Example: **LET**

Reserved word **LET**.

[Symbolic List]
The Symbolic List is a display of all the symbolic constants defined types in the system.

Format: *symbolic_name* *replacement_string*

Example: **pi** **"3.14159"**

[Type List]
The Type List is a display of all the base and user defined types in the system. The listing shows a types' name, typecode, storage requirements in bytes, and member names and typecodes.

Format: *type_name* **t:***typecode* **su:***storage_units* [{*member_name*:*typecode*}]

Example: **FLOAT** t:337 su:008

Base type **FLOAT** is typecode **337** and requires 8 bytes of storage.

POLAR t:338 su:016 {**mag**:337,**ang**:337}

User defined type **POLAR** is typecode **328** and requires 16 bytes of storage. Type has two members; **FLOAT** member **mag** and **FLOAT** member **ang**.

C12.5 Customizing the Startup Environment

A customized startup environment can easily be accomplished by loading a macro file at startup that includes any special settings the user wants, function definitions, etc. The translator automatically searches the current working directory at startup for a macro named "startup.mac."

Example: To start the system with the Reserved List off, ansi display on, and custom function library custom.mac loaded; create the following macro file startup.mac:

STARTUP.MAC: dump 11101 ansi LOAD "custom.mac"

CUSTOM.MAC

```
TYPEDEF POLAR {FLOAT mag, FLOAT ang}

DEFINE FLOAT pi()
  RETURN 3.14159265358979323846
END_DEFINE

DEFINE POLAR RecToPolar(FLOAT x, FLOAT y)
  POLAR r1
  r1.mag = SQRT(x^2+y^2)
  r1.ang = ATAN(y/x)
  IF(x<0)
    r1.ang = r1.ang+pi
  ENDIF
  RETURN r1
END_DEFINE
```

At the DOS prompt, run the program cp with the macro file "startup.mac" in the same directory.

```
SYMBOL TABLE LISTING:
[Variable List]
[Array List]
[Function List]
 RecToPolar   t:338 &v:276808 v: <nans> <nans> (x:337,y:337)
 pi           t:337 &v:276572 v: <nans> ( )
[Symbolic List]
[Type List]
 POLAR        t:338 su:016 {mag:337,ang:337}
 FLOAT        t:337 su:008
 INTEGER      t:336 su:004
 STRING       t:335 su:004
com> _
```

C13. Example Code

An example programming problem is presented in this section. Source code is included on disk in file "4bar.mac."

C13.1 Four bar crank mechanism

Program to calculate output crank angles in a four-bar mechanism using the Newton-Raphson method to solve Freudenstein's equation.

Example: Run the command processor and at the command line, enter:

com> **LOAD "4bar.mac"** <ENTER>

To execute the function, enter:

com> **bar4()** <ENTER>

Results of calculations are printed to the screen.

```
// Program to calculate output crank angles in a four-bar mechanism using the
// Newton-Raphson method to solve Freudenstein's equation.
//
// a             length of input crank, in
// b             length of coupler link, in
// c             length of output link, in
// d             length of fixed link, in
// delta_theta   increment of input angle, deg
// theta         value of input angle, deg
// theta_max     maximum value of input angle, deg
// R1,R2,R3      constants calculated from link lengths
// phi           value of output angle, deg and radians
// new_phi       improved value of output angle, deg and radians
// f1            f(phi)=R1*cos(theta)-R2*cos(phi)+R3-cos(theta-phi)
// f0            f'(phi)=R2*sin(phi)-sin(theta-phi)
// epsilon       accuracy check value, radians
DEFINE bar4()
  FLOAT a,b,c,d                          // variable declarations
  FLOAT delta_theta,theta,theta_max      //
  FLOAT R1,R2,R3,phi,new_phi             //
  FLOAT f1,f0,epsilon                    //
  INTEGER i                              //
  a = 1.0, b = 2.0, c = 2.0, d = 2.0     // variable initializations
  delta_theta = 5.0                      //
  theta = 0.0                            //
  theta_max = 360.0                      //
  phi = 41.0                             //
  epsilon = 0.00001                      //
  R1 = d/c                               // constant calculatons
  R3 = (d^2.0+a^2.0-b^2.0+c^2.0)/(2.0*c*a) //
  theta = theta*0.01745329               // convert angles to radians
  theta_max = theta_max*0.01745329       //
```

```
  phi = phi*0.01745329                          //
  delta_theta = delta_theta*0.01745329          //
  FOR(i=1; theta<=theta_max ; i=i+1, theta=theta+delta_theta)
    f1 = R1*COS(theta)-R2*COS(phi)+R3-COS(theta-phi)    // calc Freudenstein eq
    f0 = R2*SIN(phi)-SIN(theta-phi)                     // calc derivative
    new_phi = phi-f1/f0                                 // calc improved phi val
    WHILE((ABS(new_phi-phi)-epsilon) > 0)               // iterate calcs until phi converges
      phi = new_phi                                     // with accuracy specified
      f1 = R1*COS(theta)-R2*COS(phi)+R3-COS(theta-phi)
      f0 = R2*SIN(phi)-SIN(theta-phi)
      new_phi = phi-f1/f0
    ENDWHILE
    PRINT theta/0.01745329,",",new_phi/0.01745329
    phi = new_phi                               // approximate next output angle
  NEXT
END_DEFINE
```

C14. Advanced Diagnostics

The advanced diagnostic tools include:

outbuf	Toggle display of intermediate language output from parser on or off
dumpt	Toggle display of threaded code during function compilation on or off
til	Toggle between front and back end compiler environments.
del identifier	Delete a variable, array, function, or user defined type

These tools allow the developer to analyze the intermediate language output from the front end parser, study the compiled threaded code generated by the back end compiler, bypass the front end parser and work directly with the back end threaded interpreter/compiler, and observe destructor actions of symbol table objects.

C14.1 Intermediate Language Output

The front end recursive descent parser generates an intermediate language that's piped into the back end threaded interpreter/compiler. The **outbuf** command toggles on or off the display of the intermediate language that's generated.

Example: Set the system to ansi, dump 001000, scroll, outbuf, and load the file sqr.mac

SQR.MAC
```
DEFINE FLOAT sqr(FLOAT x)
  RETURN x^2.0
END_DEFINE
```

Run the program **cp** and at the command line enter:

com> **ansi dump 001000 scroll outbuf LOAD "sqr.mac"** <ENTER>

```
                Command Processor/Macro Compiler  Version 1.0  12/03/94

com> ansi dump 001000 scroll outbuf LOAD "sqr.mac"
Output =
Output =
Output = : sqr & 258516 & 258504 @f f 2.0 y^x !f rtn rtn? ;
SYMBOL TABLE LISTING:
[Function List]
 sqr            t:337 &v:258516 v: <nans> (x:337)
com> _
```

The intermediate language output from the parser is displayed.

Output =
Output =
Output = **: sqr & 258516 & 258504 @f f 2.0 y^x !f rtn rtn? ;**

Each command and function definition that's processed generates an output listing of the form:

Output = ***intermediate language stream***

Some commands are totally processed by the front end parser and generate no intermediate language stream. The first two output listings correspond to the **outbuf** and **LOAD** commands which are of this type.

The last output listing corresponds to the function definition **sqr**. All function definitions generate an intermediate language stream.

The intermediate language stream for the function definition can now be analyzed and interpreted as listed below:

:	set threaded interpreter to compile mode
sqr	keyword name
&	set number mode to address
258516	address of return value; push to address stack
&	set number mode to address
258504	address of parameter x; push to address stack
@f	pop address stack, fetch floating point value at that address and push to data stack
f	set number mode to floating point
2.0	floating point number; push to data stack
y^x	raise second data stack entry to power in first stack entry, pop pop data stack, push value to data stack
!f	store floating point number on data stack at address on address stack, pop data and address stacks
rtn	function return
rtn?	check function has a return
;	terminate keyword definition; set threaded interpreter to execute mode

Note: The Symbol Table Listing can be set to local mode to verify the address of parameter x using the **LOCAL** command.

C14.2 Threaded Code Generation

The back end threaded interpreter/compiler processes the intermediate language stream piped to it by the front end recursive descent parser. Commands are interpreted and executed and produce no threaded code. Functions are compiled and produce a fully analyzed threaded code. The **dumpt** command toggles on or off the display of the threaded code that's generated when a function is compiled.

Example: Set the system to ansi, dump 001000, scroll, dumpt, outbuf, and load the file sqr.mac

```
SQR.MAC     DEFINE FLOAT sqr(FLOAT x)
              RETURN x^2.0
            END_DEFINE
```

Run the program **cp** and at the command line enter:

com> **ansi dump 001000 scroll dumpt outbuf LOAD "sqr.mac"** <ENTER>

The threaded code produced by the front end compiler is displayed as shown below.

```
                Command Processor/Macro Compiler  Version 1.0  12/03/94

com> ansi dump 001000 scroll dumpt outbuf LOAD "sqr.mac"
Output =
Output =
Output = : sqr & 258516 & 258504 @f f 2.0 y^x !f rtn rtn? ;
[172676] 258528 <sqr>
[172680] 172664
[172684] 3534
[172688] 171312 <adr_lh>
[172692] 258516
[172696] 171312 <adr_lh>
[172700] 258504
[172704] 171772 <@f>
[172708] 171300 <float_lh>
[172712] 258536
[172716] 172672 <y^x>
[172720] 171820 <!f>
[172724] 171296 <semi>
SYMBOL TABLE LISTING:
[Function List]
 sqr            t:337 &v:258516 v:nans (x:337)
com> _
```

The threaded code listing can now be analyzed and interpreted as listed below:

[172676] 258528 <**sqr**>	address of keyword lexeme "sqr"
[172680] 172664	link address of next keyword in dictionary
[172684] 3534	address of inner interpreter routine "colon"
[172688] 171312 <adr_lh>	address of address literal handler
[172692] 258516	address of function return value
[172696] 171312 <adr_lh>	address of address literal handler
[172700] 258504	address of parameter "x"
[172704] 171772 <@f>	address of floating point fetch operator
[172708] 171300 <float_lh>	address of floating point literal handler
[172712] 258536	address of floating point constant 2.0
[172716] 172672 <y^x>	address of routine "y^x"
[172720] 171820 <!f>	address of floating point store operator
[172724] 171296 <semi>	address of inner interpreter routine "semi"

The addresses in brackets are actual memory addresses where the threaded code is stored. The threaded code is actually a list of addresses. The mnemonic to the right of the threaded code entries identify the routines associated with the addresses.

Note: The Symbol Table Listing can be set to local mode to verify the address of parameter x using the **LOCAL** command.

C14.3 Back End Threaded Interpreter/Compiler

The back end threaded interpreter/compiler environment can be entered bypassing the front end recursive decent parser by using the command **til**. This command toggles between environments.

Example: Run the program **cp** and at the command line enter:

com> **LOAD "sqr.mac"** <ENTER>
com> **til** <ENTER>

The system is now in the back end compiler environment. At the command line enter:

com> **list** <ENTER>

list is a keyword in the threaded interpreter/compiler language that lists the TIL system dictionary to the screen as shown below.

```
com> list
---------------------------------------- CORE VOCABULARY ----------------------------------------
$sqr      global    local     del       sys       trans     ftoa      inps      inpi      inpf
input     closea    close     open      cprt      fprt      fprts     fprti     fprtf     prt
prts      prti      prtf      mem       y^x       abs       sqrt      log       ln        exp
tanh      cosh      sinh      atan      acos      asin      tan       cos       sin       Y|X
y|x       Y&X       y&x       Y<>X      y<>x      Y=X       y=x       Y>=X      y>=x      Y>X
y>x       Y<=X      y<=x      Y<X       y<x       !X        !x        x<>0      NEG       neg
s+        i-        i+        i*        -         +         /         *         acpy      OF+
of+       aiof      aof       @&vd      @&va      !&v       !&S+      !&S       !&s+      !&s
@&S+      @&S       @&s+      @&s       !I+       !I        !i+       !i        @I+       @I
@i+       @i        !F+       !F        !f+       !f        @F+       @F        @f+       @f
i2>f      i>f       list      :
---------------------------------------- IMMEDIATE VOCABULARY ----------------------------------------
&v        &s        f         ii        i         &         rtn?      rtn       dump
---------------------------------------- COMPILER VOCABULARY ----------------------------------------
cjmp      default   SEND      send      CASE3     case3     case2     CASE1     case1     SWTCH
swtch     setbrk    break     next      for2      for1      for       until     repeat    wend
while     jmp       endif     elseif    if        ;
com> _
```

Note the function **sqr** that was compiled earlier is now a new keyword in the language and appears as the first entry in the CORE VOCABULARY section of the listing as **$sqr**.

At this point, the developer can interact with the TIL but must have knowledge of storage addresses to do anything useful. Toggling back to the front end parser, addresses can be retrieved from the Symbol Table Listing.

Example: To call the function sqr with a parameter value of 6.0 and return value address of 258504, at the command line enter:

com> **& 258504 f 6.0 !f $sqr** <ENTER>

Now toggle back to the front end parser and inspect the contents of the return value for the function sqr.

At the command line enter:

com> **til** <ENTER>

Hit <ENTER> again to display the Symbol Table Listing shown below. Note the current return value of the function sqr is equal to 36.0.

```
SYMBOL TABLE LISTING:
[Variable List]
[Array List]
[Function List]
 sqr            t:337 &v:258504 v: <36> (x:337)
[Reserved List]
 LET            TYPEDEF     DEFINE      END_DEFINE  RETURN      EXTERN
 IF             ELSEIF      ELSE        ENDIF       FOR         NEXT
 WHILE          ENDWHILE    REPEAT      UNTIL       SWITCH      CASE
 DEFAULT        ENDSWITCH   BREAK       INPUT       PRINT       OPEN
 CLOSE          LOAD        FTOA        TRANSLATE   DELETE      SYMBOL
 SYSTEM         NOT         AND         OR          SIN         COS
 TAN            ASIN        ACOS        ATAN        SINH        COSH
 TANH           ABS         SQRT        LOG         LN          EXP
 LOCAL          GLOBAL      dump        scroll      outbuf      dumpt
 til            ansi        destruct    clock       mem         exit
[Symbolic List]
[Type List]
 FLOAT          t:337 su:008
 INTEGER        t:336 su:004
 STRING         t:335 su:004
com> _
```

C14.4 Symbol Table Object Destructors

Symbol Table records are actually "C++" objects with constructors and destructors. To insure that destructors release allocated memory, deletion messages are currently listed to the screen whenever an object is deleted. The **DELETE** command is used to delete a variable, array, function, symbolic constant, or user defined type.

Syntax: **DELETE** "identifier"

Example: Run the program **cp** and at the command line enter:

com> **ansi dump 000001 scroll** <ENTER>
com> **TYPEDEF POINT { FLOAT x, FLOAT y }** <ENTER>
com> **scroll** <ENTER>

Now delete the user defined type POINT by entering the following:

com> **DELETE "POINT"** <ENTER>

The type object destructor messages appear as shown below:

```
com> TYPEDEF POINT { FLOAT x, FLOAT y }
SYMBOL TABLE LISTING:
[Type List]
 POINT        t:338 su:016 {x:337,y:337}
 FLOAT        t:337 su:008
 INTEGER      t:336 su:004
 STRING       t:335 su:004
com> DELETE "POINT"
Deleting type record POINT
...deleting lexeme string
...deleting storage offset list
...deleting subtype list
 Deleting subtype record y
 ...deleting lexeme string
 Deletion complete
 Deleting subtype record x
 ...deleting lexeme string
 Deletion complete
Deletion complete
SYMBOL TABLE LISTING:
[Type List]
 FLOAT        t:337 su:008
 INTEGER      t:336 su:004
 STRING       t:335 su:004
com> _
```

Note: If a user defined type is deleted when an identifier of that type exists, a system crash will occur. Delete all identifiers of a specific user defined type before deleting the type. If a function exists that uses the type, delete the function first.

C15. Error Messages

C15.1 Lexical

End of stream reached before comment block was closed
A comment block was started with "/*" but never closed with "*/".
End of stream reached before string literal block was closed
A string literal was started with "" but never closed with "".
Illegal character in ASCII range 128-255
Command stream contained an illegal character; any character above ASCII 127 is illegal.
Missing exponent in number
Floating point number is missing exponent, e.g., 3.5E- or 3.5E+.
Token exceeds maximum character length of MaxTokenSize = *integer value*
A token was entered that exceeds the maximum system token size.

C15.2 Parser

5 digit binary number expected
System expected a 5 digit binary number composed of 0's and 1's; e.g., 00101
Array index is not an INTEGER
Expression or value used to specify an array indice was not an INTEGER.
Array parameter missing closing]
An array parameter in a function definition is missing a closing]; e.g., DEFINE test(FLOAT vect[ ).
Attempt to assign a value to a function
Function was used as a lvalue in an assignment statement; e.g., myfunc()=3. This type of error usually generates the error message "Syntax error in COMMAND module", so this message may never appear.
Attempt to use a function that has no return value in an expression.
A function with no return value was used in an expression expecting a value.
BREAK statement cannot be used outside of a FOR, WHILE, REPEAT, or SWITCH block
A BREAK statement occurred outside of an iterative or SWITCH block.
Cannot open file
File could not be opened using the LOAD command.
Cannot open startup file
File could not be opened when starting program from DOS prompt with a file parameter.
CASE expression type mismatch
CASE expression type is not the same as SWITCH expression and could not be implicitly cast to match
Function does not exist
Attempt to set the Symbol Table Listing to local mode for a function that does not exist.
Function has no return value
A function was defined to return a value but did not include a RETURN *value* statement in the function body.
Identifier has not been declared
Attempt to use an identifier that was never declared.
Illegal CASE expression type
A CASE expression returns a type that is not allowed in a SWITCH or CASE expression.
Illegal filename
An illegal filename was used with the LOAD command.
Illegal or missing identifier
Syntax was expecting a valid identifier.

Illegal or missing macro identifier
Attempt to define a function with an illegal or missing function identifier.

Illegal parameter identifier
An illegal identifier was used in naming a function parameter.

Illegal reference parameter
A function was called with an illegal reference parameter.

Illegal SWITCH expression type
A SWITCH expression returns a type that is not allowed in a SWITCH or CASE expression.

Incorrect number of array indices specified
An array used in an assignment or expression does not have the correct number of indices to match its dimensions.

Incorrect number of function parameters
Attempt to call a function with an incorrect number of parameters.

Number of array dimensions exceeds maximum limit of MaxArrayDims = *integer value*
Attempt to declare a multidimensional array that exceeds the system dimension limit.

Operator * undefined for current operand(s) type
Operator + undefined for current operand(s) type
Operator - undefined for current operand(s) type
Operator / undefined for current operand(s) type
Operator < undefined for current operand(s) type
Operator <= undefined for current operand(s) type
Operator <> undefined for current operand(s) type
Operator = undefined for current operand(s) type
Operator > undefined for current operand(s) type
Operator >= undefined for current operand(s) type
Operator ^ undefined for current operand(s) type
Operator AND undefined for current operand(s) type
Operator OR undefined for current operand(s) type
Operator unary + undefined for current operand type
Operator unary - undefined for current operand type
Operator unary NOT undefined for current operand type
Attempt to use an operator that is not defined for the operand(s) type; e.g., applying the unary - to a string variable.

Parameter mismatch
Attempt to call a function with a parameter that does not match the function's parameter type.

Parameter previously declared
Attempt to use a duplicate parameter identifier in a function definition.

Subidentifier does not exist
Attempt to access a nonexistent member of a user defined type.

Subidentifier previously declared
Attempt to duplicate a member identifier in a TYPEDEF statement.

SWITCH statement must have at least one CASE statement
A SWITCH statement requires at least one CASE statement.

Syntax error in COMMAND module
Command syntax error. Syntax was expecting the start of a declaration, assignment, or typedef statement or a function call.

Syntax error in MATCH module
Grammatical syntax error. Most general syntax error that occurs when an expected element of the grammar is missing; e.g., An expected matching operator such as], }, or) or keyword such as UNTIL, NEXT, ENDWHILE, END_DEFINE, etc. Missing an = operator in an assignment expression. Missing a (or { in a function or typedef definition, etc.

Syntax error in PRIMARY module
Expression syntax error. Syntax was expecting a valid expression operand, system function identifier, or parenthesis.

Token is not a legal subidentifier
Attempt to use an illegal identifier to access a member of a user defined type.
Token is not a legal subtype identifier
Attempt to use an illegal member identifier in a TYPEDEF statement.
Unknown type specifier
Attempt to apply an undefined type specifier on a member in a TYPEDEF statement.
Unknown/missing parameter type specifier
Missing or undefined type specifier in function definition parameter list.
Upper array index must be >= to lower array index
Attempt to declare an array with a starting indice value higher than its ending indice value; e.g., FLOAT array[3:2].

C15.3 Symbol Table

Attempt to delete a base type
Base types cannot be deleted.
Attempt to delete a reserved word
Reserved words cannot be deleted.
Attempt to delete unknown record type
Attempt to delete something that is not a symbol table record; e.g., trying to delete a number or nonexistent identifier.
External array does not exist
Attempt to declare a nonexistent EXTERN array in a function definition.
External variable does not exist
Attempt to declare a nonexistent EXTERN variable in a function definition.
Memory allocation failure
Probable attempt by system to allocate memory for a new symbol table entry has failed. Memory available to the system is low or a large request was made; e.g., declaring a very large array. The system also temporarily allocates memory in many different areas and a failure would generate this error message. Also, the system would generate this message at startup if insufficient memory existed to initialize the system.
Type mismatch
An assignment or expression involving incompatible types.
Type records can only be deleted LIFO
Type records must be deleted in order starting with the last record created. This prevents a user defined type from being deleted that's a member's type of another user defined type.

C15.4 Compiler

TIL dictionary space is full
There is not enough room left in the TIL dictionary space to compile the current function. Once the dictionary becomes full, no more functions can be compiled. Exit and restart the command processor to purge the dictionary; reload only necessary functions to reduce dictionary space required.

C15.5 Runtime

Array bounds exceeded
Attempt to access an array outside of its dimension bounds.

Floating Point Exception
Floating point exception error has occurred; e.g., division by 0.

Function structure caused a return with no value
A function that returns a value was executed but did not return a value. This happens when a function's structure is such that it can execute to completion but bypass all its RETURN statements.

Incorrect number of array indices specified
An array was passed to a function that does not have the correct number of indices.

C15.6 System

System errors are generated to trap system programming errors and help the developer locate and debug program code where the error originated. The user cannot resolve these types or errors; the problem is in the system code. Hopefully, no system errors will ever be generated. However, if a system error does occur, it should be reported to the developer.

Array parameters cannot be call by value
Symbol table attempted to create an array function parameter as call by value.

Emitter error
Emitter module could not recognize the type of token sent to it to emit.

Illegal base type in storage offset list
Emitter module retrieved a storage offset list from the symbol table that contained a nonbase type.

Illegal/unknown parameter record type
Symbol table failed to recognize a parameter record type when attempting to insert a parameter in the symbol table.

Parameter record not found
Symbol table failed to locate a function parameter record.

Scope resolution error
Symbol table failed to set scope of variable or array Symbol Table Listing when using local or global command.

Stack Underflow
A stack underflow has occurred in the back end compiler.

Type stack under/over flow
A type stack underflow or overflow has occurred in the parser.

Unknown BASE data type detected
Symbol table detected an unknown base data type when retrieving an address contents for display in the Symbol Table Listing.

Unknown BASE data type detected during storage initialization
Symbol table detected an unknown base data type when initializing a storage block.

Unknown parameter record
The parser failed to retrieve a function's parameter record from the symbol table during the parameter assignment phase of a function call.

Unknown record scope
Symbol table attempted to insert or return a record with an unknown scope.

Unknown storage TYPE
Symbol table failed to allocate storage for an entity because its type record could not be found.

Unknown token class detected in scanner module
State table in scanner module failed to classify a token.

BIBLIOGRAPHY

Aho, Alfred V., John E. Hopcroft, and Jeffery D. Ullman [1974]. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass.

Aho, Alfred V., Ravi Sethi, and Jeffery D. Ullman [1986]. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, Mass.

Akin, J. Ed [1990]. *Computer-Assisted Mechanical Design*, Prentice Hall, Englewood Cliffs, NJ.

Burden, Richard L., and J. Douglas Faires [1985]. *Numerical Analysis*, Prindle, Weber, and Schmidt, Boston.

Calingaert, Peter [1979]. *Assemblers, Compilers, and Program Translation*, Computer Science Press, Inc., Rockville, MD.

Cannon Jr., Robert H. [1967]. *Dynamics of Physical Systems*, McGraw-Hill, Inc., New York, NY.

Coan, James S. [1978]. *Basic BASIC: An Introduction to Computer Programming in BASIC Language*, Hayden Book Company, Inc., Rochelle Park, NJ.

Ellis, T. M. R. [1982]. *A Structured Approach to Fortran 77 Programming*, Addison-Wesley Publishers Limited, London.

Hewlett-Packard [1989]. *ME10d Mechanical Engineering CAD System - Writing Macros Manual*, Federal Republic of Germany.

Horowitz, Ellis [1984]. *Fundamentals of Programming Languages*, Computer Science Press, Inc., Rockville, MD.

James, M. L., G. M. Smith, and J. C. Wolford [1985]. *Applied Numerical Methods For Digital Computation*, Harper & Row, Publishers, Inc., New York, NY.

Kernighan, Brian W., and Dennis M. Ritchie [1988]. *The C Programming Language*, Prentice Hall, Englewood Cliffs, NJ.

Lewis, P. M., II, D. J. Rosenkrantz, and R. E. Stearns [1976]. *Compiler Design Theory*, Addison-Wesley, Reading, Mass.

Loeliger, R. G. [1981]. *Threaded Interpretive Languages*, Byte Publications, Inc., Peterborough, NH.

Microsoft [1991]. *C Language Reference*, Microsoft Corporation, Redmond, WA.

Microsoft [1991]. *C++ Language Reference*, Microsoft Corporation, Redmond, WA.

Microsoft [1991]. *C++ Tutorial*, Microsoft Corporation, Redmond, WA.

Purdum, Jack J. [1985]. *C Programming Guide*, Que Corporation, Indianapolis, IN.

Schildt, Herbert [1992]. *Teach Yourself C++*, McGraw-Hill, Inc., Berkeley, CA.

Speckhart, Frank H., and Walter L. Green [1976]. *A Guide To Using CSMP - The Continuous System Modeling Program*, Prentice-Hall, Inc., Englewood Cliffs, NJ.

Stroustrup, Bjarne [1986]. *The C++ Programming Language*, Addison-Wesley, Reading, Mass.

Symantec [1991]. *C++ Video Course*, Symantec Corporation, Cupertino, CA.

Symantec [1991]. *Zortech C++ Compiler Guide*, Symantec Corporation, Cupertino, CA.

Symantec [1991]. *Zortech C++ Function Reference*, Symantec Corporation, Cupertino, CA.

Symantec [1991]. *Zortech C++ Numerics Programming Guide*, Symantec Corporation, Cupertino, CA.

Symantec [1993]. *Symantec C++ Professional Compiler and Tools Guide*, Symantec Corporation, Cupertino, CA.

Symantec [1993]. *Symantec C++ Run-Time Library Reference*, Symantec Corporation, Cupertino, CA.

Symantec [1993]. *Symantec C++ User's Guide and Reference*, Symantec Corporation, Cupertino, CA.

Thomson, William T. [1981]. *Theory of Vibration with Applications*, Prentice Hall, Englewood Cliffs, NJ.