An implementation of four of Ledgard's mini-languages

Piyanai Saowarattitada

Follow this and additional works at: http://scholarworks.rit.edu/theses

Recommended Citation

This Thesis is brought to you for free and open access by the Thesis/Dissertation Collections at RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
An Implementation of Four of Ledgard's Mini-languages

by
Fiyanai Saowarattitada

A thesis, submitted to
The Faculty of the School of Computer Science and Technology, in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

by
Fiyanai Saowarattitada

A thesis, submitted to
The Faculty of the School of Computer Science and Technology, in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

June 29, 1983
Title of Thesis: An Implementation of Four of Iedgard's Mini-languages

I ____________________________ hereby (grant/deny) permission
to the Wallace Memorial, of RIT, to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

or

I Piyana Saowarattitada prefer to be contacted each time a request for reproduction is made. I can be reached at the following address:

16 Kimball Drive
Rochester, N.Y. 14623

Date: July 13, 1983
ACKNOWLEDGEMENTS

I would like to thank Dr. John I. Ellis for spending so much time helping me with this thesis. I also would like to thank John A. Piles and Chris Comte for being members of my committee, Dr. Lawrence Coon for giving me some ideas for the symbol table management for Mini-language Type.

Special thanks to Michael G. Unger for encouraging me to go through all the problems and for correcting my poor American English. Without him I would have had to spend much more time completing this thesis.
For decades humans have been searching for the best way to communicate with an intelligent machine, a computer. Several programming languages have been written with the idea of a universal language which includes solutions to solve as many problems as one can think of. But the more universal the languages are the more complex they are to study.

Henry Iedgara tried to combine these two ideas together. He suggests the idea of studying programming language by dealing with a few key features at a time. He separated the various programming features, grouped the similar ones together and wrote his own small languages called "Mini-languages".

"The programming landscape" (15) which includes 13 Mini-languages was used as the central reference for this thesis work. Each of the four Mini-languages was implemented in 2 sections; a compiler and an interpreter. One can write a program in any of the four Mini-languages; compile and run (interpret) it to test the correctness.

Key words: Mini-language, compiler, interpreter, lexical
analysis, syntax analysis, Iex, Yacc, intermediate code, quadruples, code optimization, code generation, three address code, symbol table, hash function, error handling, grammar rules, C structure, iterative statement, pass by value, pass by result, pass by value_result, pass by location (reference) IAIR(1), syntax-direct translation, postfix form, syntax tree, trile, C programming language, regular expression, token, VAX 11/780, UNIX.
# Table of Contents

1. Overview ................................................................................. 1  
   1.1 Mini-language ................................................................. 1  
   1.2 Language Design Goals .................................................. 2  
   1.3 Language Tools ............................................................... 3  
   1.4 Host System .................................................................... 4  

2. Compilers ................................................................................ 5  
   2.1 Lex - A Lexical Analyzer Generator ............................... 9  
      2.1.1 Lex Source ............................................................... 10  
   2.2 Yacc - Yet Another Compiler-Compiler ....................... 12  
      2.2.1 Specifications .......................................................... 13  
      2.2.2 Yacc Parser ............................................................ 15  
      2.2.3 Error Handling in Yacc ......................................... 18  
      2.2.4 Lex with Yacc ......................................................... 19  
   2.3 Intermediate Code Generation ....................................... 21  
      2.3.1 Syntax-Directed Translation .................................... 23  
      2.3.2 Control-Flow Representation .................................. 25  
   2.4 Interpreter .................................................................... 27  

3. Mini-Languages ....................................................................... 31  
   3.1 Mini-language Core ....................................................... 31  
      3.1.1 Symbol Table .......................................................... 31  
      3.1.2 Semantic of Mini-language Core ............................ 32  
      3.1.3 Declaration ............................................................. 33  
      3.1.4 Assignment Statement ........................................... 34
7. USER MANUAL ................................................. 181
    7.1 Additional Comment ...................................... 183
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Phases of a compiler</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Example Yacc for Mini-language Core</td>
<td>14</td>
</tr>
<tr>
<td>2.3</td>
<td>Example of Yacc (in human readable form) generated by invoking flag <code>-v</code></td>
<td>15</td>
</tr>
<tr>
<td>2.4</td>
<td>Comparison of each type of intermediate code</td>
<td>22</td>
</tr>
<tr>
<td>2.5</td>
<td>Quadruple representation of three-address statements</td>
<td>23</td>
</tr>
<tr>
<td>2.6</td>
<td>An Example of grammars rules in Mini-language Core</td>
<td>22</td>
</tr>
<tr>
<td>2.7</td>
<td>Trace of syntax-directed translation</td>
<td>25</td>
</tr>
<tr>
<td>2.8</td>
<td>Form of code for constructs using Boolean expressions</td>
<td>26</td>
</tr>
<tr>
<td>3.1</td>
<td>Format of Mini-language</td>
<td>25</td>
</tr>
<tr>
<td>3.2</td>
<td>A nested-if in Mini-language Core</td>
<td>26</td>
</tr>
<tr>
<td>3.3</td>
<td>Backpatching corresponding to the nesting if in fig 3.2</td>
<td>36</td>
</tr>
<tr>
<td>3.4</td>
<td>Intermediate code for Mini-language Core</td>
<td>37</td>
</tr>
<tr>
<td>3.5</td>
<td>Intermediate code for while statement</td>
<td>36</td>
</tr>
<tr>
<td>3.6</td>
<td>An Example of a program that is not a D-structure</td>
<td>46</td>
</tr>
<tr>
<td>3.7</td>
<td>A Mini-language D program</td>
<td>47</td>
</tr>
<tr>
<td>3.8</td>
<td>Example of quadruple for array type in Mini-language Type</td>
<td>57</td>
</tr>
<tr>
<td>3.9</td>
<td>Example of a record structure declaration of record type in Mini-language Type</td>
<td>58</td>
</tr>
<tr>
<td>C.1</td>
<td>Structure used in each implemented Mini-language</td>
<td>104</td>
</tr>
</tbody>
</table>
CHAPTER 1

OVERVIEW

The programmer's most important tool is, of course, a programming language. A good language can lead the programmer to the correct solution of a problem in a natural and easy manner. Conversely, a poor language may add so much complexity to finding the solution that the programmer will abandon the attempt at solving the original problem in favor of an easier one.

1.1 Mini-languages

Mini-languages are examples of good languages, each of which has been designed around some key language features. They allow a concept to be studied without the need to understand the detail and complexity found in real programming languages such as Pascal or Cobol. In another words, a Mini-language isolates control structures to concentrate on one group (subset) of programming features at a time. Henry Iedgard used the simplicity of Mini-languages to introduce ideas dealing with the studying of programming languages (15).

There are several approaches to the study of programming languages. One is to examine several existin...
languages in detail, compare and contrast their salient features, and attempt to draw conclusions about underlying design principles. Another interesting approach starts with the design principles, studies them in relative isolation, and then seeks examples of the implementation of these principles in real languages. The use of Mini-languages is central to the second approach. One of the first uses of this technique was in Iedgard's paper, "Ten Mini-languages: A study of Topic Issues in Programming Languages" (14). In 1980 Iedgard went another step in using Mini-languages. He and Michael Marcotty wrote "The Programming Language Landscape" which includes 12 useful Mini-languages which reflect the work done by Iedgard on the design of Ada. Most of the languages are built on a common core, as shown in a Mini-language 'Core'.

1.2 Language Design Goals

Implementing 4 of Iedgard's Mini-languages is the goal of this project. The first is the basic Mini-language 'Core'. The second is the Mini-language 'D' which concentrates on I-structures. D for Dijkstra [as in Pruso and Steiglitz 1972]. A D-structure is a one-in, one-out structure, and no transfer of control can occur during its execution. The Mini-language 'Type' is the third language. In this language the composite types are arrays of a given simple type (integer, string, boolean) and record structures.
The last is the *Miri-language Procedure* which concentrates on the ways in which the arguments are passed from the calling procedure to the called procedure. It includes pass by value, pass by result, pass by value_result and pass by location (or reference).

There are two sections in each implementation. The first is a one pass compiler which includes three phases, lexical analysis, parsing and intermediate code generation. The second is an interpreter which insures the correctness and consistency of each compiler.

1.3 Language Tools

Yacc and Lex are UNIX tools used in implementing compilers for each of the 4 languages. Yacc (*Yet Another Compiler-Compiler*) (12) is a parser generator. In using Yacc a user will specify the structures of the input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a C function that parses the input. The class of specifications accepted is a very general one: a context-free grammar with disambiguating rules.

Lex (*Lexical Analyzer Generator*) (16) is a program generator designed for lexical processing of character input streams. It accepts a high-level problem oriented specification for character string matching and produces a program in
a general purpose language which recognizes regular expressions written by the user himself. Lex turns the user's expressions and actions (called source) into program segments in the host general-purpose language: the generated procedure is named `yylex',.

It is particularly easy to interface Lex and Yacc. The generated Lex program recognizes only regular expressions; Yacc generates parsers that accept a language in the class of context free grammars, but which require a lower level analyzer to recognize input tokens. When used as a preprocessor for a later parser generator, Lex is used to partition the input stream, and Yacc assigns a structure to the resulting pieces.

1.4 Host System

All work for this implementation was done on a VAX 11/780 system using the UNIX operating system. UNIX is chosen because it provides an excellent environment in which to develop software due to its many useful utility programs. Some tools provided by the system such as Lex (Lexical Analyzer Generator) and Yacc (Yet Another Compiler-Compiler) were used throughout this project. The C programming language was used for the coding part, since C is the most suitable language for the UNIX system.
A compiler takes as input a source program and produces as output an equivalent sequence of machine instructions. It is much simpler to consider a compiler in subprocesses called phases as shown in Fig 2.1. A phase is a logically cohesive operation that takes as input one representation of the source program and produces as output another representation.

In the first phase, the lexical analyzer, or scanner, separates characters of the source language into groups that logically belong together; these groups are called tokens. Keywords such as IF or WHILE, operator symbols such as -, or +, identifiers such as X or NUMBER and punctuation symbols such as parentheses or commas are examples of tokens. The output of the lexical analyzer is a stream of tokens which is passed to the next phase, the syntax analyzer or parser. The tokens in this stream can be represented mostly by integer codes.

The intermediate code generator, the third phase, uses the structure produced by the syntax analyzer to create a stream of simple instructions. Three-address code (quadruples) is the intermediate code used in many compilers, since
it can be easily rearranged.

Code optimization is an optional phase designed to improve the intermediate code so that the ultimate object program runs faster and/or takes less space. Its output is another intermediate code program that is equivalent to the original code and may save space or time.

The last phase is code generation, which produces the object code by deciding on the memory locations for data, selecting code to access each datum and selecting the registers in which each computation is to be done.

The table-management or bookkeeping portion keeps track of names used by the program and records essential information about each, for example its type (integer, string, etc.). The data structure used to record this information is called a symbol table. Several access techniques can be used. A linear search is the simplest way but has the trade off of consuming extra time and space. The most popular technique is to use hashing which save time and space. For this implementation, a hash table for each Mini-language was generated.

The error handler is invoked when a flaw in the source program is detected. A warning message will be given for each error detected. It is desirable that compilation be completed or flawed programs, at least through the syntax-
analysis phase. In general when the compiler comes to a point in the input stream where it cannot continue processing a valid phase, some possible changes will be produced by the compiler in order to continue detecting the program. Some possible changes are:

(1) Alteration of a single character. For example, if the parser is given the identifier INTEGER by the lexical analyzer and it is not proper for an identifier to appear at this point in the program, the parser may guess that the keyword INTEGER was meant.

(2) Insertion of a single token. For example, the parser can replace 2C by 2*C.

(3) Deletion of a single token. For example, a comma might be inserted incorrectly after the 10 in a FORTRAN statement such as DC 10, I = 1.20.

The table-management and the error handling routine both interact with all phases of the compiler as shown in Fig 2.1.
For each implementation of a Mini-language, Lexical analysis was handled by 'Lex' and 'Yacc' was used for the syntax analysis phase. The intermediate code for each Mini-language was also generated using three-address code. This intermediate code was in the form of an array of quadruples.
However instead of implementing code generation for each Mini-language, an interpreter was written and used to run the intermediate code and thereby test the correctness and completeness of each compiler.

2.1 IEX - A LEXICAL ANALYZER GENERATOR

Iex is one of the tools used to help in lexical analysis of text. Iex is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages called "host languages". Just as general purpose languages can produce code to run on different computer hardware, Iex can write code in different host languages. At present, there is only one host language, C, which was used in this implementation.

The Lex source is a table of regular expressions and corresponding program fragments. The table is translated into a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized, the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Iex.

Lexical analysis programs written with Iex accept ambiguous specifications and choose the longest match possible
at each input point. If necessary substantial lockehead is performed on the input, but the input stream will be backed up to the end of the current partition.

### 2.1.1 Lex Source

The general format of Lex source is as follows:

```plaintext
{definitions} : can be omitted.
%% : required.
{rules} : can be omitted.
%% : can be omitted.
{user subroutines} : can be omitted.
```

The absolute minimum Lex program (in theory) is

```plaintext
%%
```

(no definition, no rules) which translates into a program which copies the input to the output unchanged.

Rules represent the user's control decisions. Each rule has the form

```
regular expression actions
```

A regular expression specifies a set of strings to be matched. The letters of the alphabet and digits are always text characters. For example, the regular expression

```
program
```

matches the string "program" whenever it appears.
expressions in lex use the following operators:

\begin{align*}
\text{x} & \quad \text{the character \textquoteleft x\textquoteright } \\
x^" & \quad \text{an \textquoteleft x\textquoteright , even if \textit{x} is an operator. } \\
\backslash x & \quad \text{an \textquoteleft x\textquoteright , even if \textit{x} is an operator. } \\
[xy] & \quad \text{the characters \textit{x},\textit{y} or \textit{z}. } \\
[x-z] & \quad \text{the characters \textit{x},\textit{y} or \textit{z}. } \\
["x] & \quad \text{any character but \textit{x}. } \\
_. & \quad \text{any character but newline. } \\
x & \quad \text{an \textit{x} at the beginning of a line. } \\
<y>x & \quad \text{an \textit{x} when \textit{lex} is in condition \textit{y}. } \\
x^* & \quad \text{an \textit{x} at the end of a line. } \\
x? & \quad \text{an \textit{x} at the beginning of a line. } \\
x^0,1,2,... & \quad \text{instances of \textit{x}. } \\
x^1,2,3,... & \quad \text{instances of \textit{x}. } \\
x|y & \quad \text{an \textit{x} or a \textit{y}. } \\
(x) & \quad \text{an \textit{x}. } \\
x/y & \quad \text{an \textit{x} but only if followed by \textit{y}. } \\
\{xx\} & \quad \text{the translation of \textit{xx} from the definitions section. } \\
\text{x\{m,n\}} & \quad \text{\textit{m} through \textit{n} occurrences of \textit{x}. }
\end{align*}

for example, the rule

\begin{verbatim}
program printf: found keyword program
\end{verbatim}

means look for the string \texttt{program} in the input stream and print the message \texttt{\textquoteleft found keyword program\textquoteright } whenever it appears. The end of the regular expression is indicated by the first blank or tab character. Traces are used when there is more than one line of action (compound).

\textit{Lex} has the ability to handle ambiguous specifications. Some rules are applied when more than one expression matches the current input. These rules are

(1) The longest match is preferred.
Among lexical rules which match the same number of characters, the rule given first is preferred. For example, suppose the rules

```
program [a-z]+ keyword actions..;

id [a-z]+ identifier actions.;
```

are given in this order. If the input is "programs", it is taken as a keyword as the first matching rule is preferred. Anything shorter than "program" will not match the expression "program" and so the identifier interpretation is used.

Lex normally partitions the input scheme, it does not search for all possible matches of each expression. This means that each character is accounted for once and only once. Actions are program fragments to be executed when the expressions are recognized.

2.2 YACC - YET ANOTHER COMPILER-COMPILER

Yacc provides a general tool for imposing structure on the input to a computer program. It converts a context-free grammar into a set of tables for a simple automaton which executes an 1R1 parsing algorithm. If the grammar is ambiguous, precedence rules may be specified to resolve the ambiguity.
The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to perform the basic input. Yacc then generates a function to control the input process. This function, called a parser, calls the user-supplied low-level input routine (the lexical analyzer) to pick up tokens from the input stream. These tokens are organized according to the input grammar rules. When one of these rules has been recognized, an action (user code supplied) is invoked. The actions have the ability to return values and make use of the values of other actions.

2.2.1 Specifications

The basic specification file consists of three sections separated by "%%". The general form is as follows:

```
declarations
%%
rules
%%
program
```

: may be empty.

: can be omitted.

The smallest legal Yacc specification is

```
%%
rules
```

The rules section consists of one or more grammar rules. Each rule has an associated action which is to be
performed each time the rule is recognized in the input process. These actions may return values and may use the values returned by previous actions.

An action is just a group of C statements. '{' and '}' are used when there is more than one C statement in an action. Fig. 2.2 shows a part of Mini-language Core to give a general idea of Yacc:

```
program :  PFCGRAM
  declareseq
  content
  |  error ES
  |
  |  /* if 'program' keyword is missing,**/
  |  /* print out the error message */
  |  usage( Missing PROGRAM keyword 0 );
  |
  declareseq content
  |
  |  PROGRAM error ES
  |
  |  usage( What is next ?' );
  |
  |
  declareseq :  declare
  |
  |  declareseq declare
  |
  |
  declare :  IFCIAHE iderlist ES
  |
  |
  idenlist :  iderlist IDENTIFIER IDENTIFIER
  |  IDENTITY
  |
  |
  .... next rules ...
```

Fig 2.2 Example Yacc for Mini-language Core.
As in the example, '|' can be used to avoid rewriting the left hand side when there are several grammar rules with the same left hand side. The rule

\[
\text{declare} : \quad \text{DECLARE idenlist FS}
\]

means that this rule will be reduced to 'declare' when Yacc found a key word DECLARE, followed by a list of identifiers (a rule idenlist') and the semicolon ('ES'). The algorithm used by the Yacc parser encourages 'left recursive' grammar rules. This is the reason why 'left recursive' rules are used in all the grammar rules of this implementation. For example in the above example:

\[
\text{idenlist} : \quad \text{idenlist IDENTIFIER'} \\
\text{idenlist IDENTIFIER}
\]

2.2.2 Yacc Parser

The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token, the lookahead token. The current state is always the one on top of the stack. The states of the finite state machine are given small integer labels. Initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.
Shift, reduce, accept and error are 4 actions available for the machine. A move of the parser is done as follows:

1) Pased or its current state, the parser decides whether it needs a lookahead token to decide what action should be done. If it needs one, and does not have one, it calls "yylex" (lex program) to obtain the next token.

2) Using the current state, and the lookahead token, if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack or popped off of the stack, and the lookahead token being processed or left alone. When Yacc is invoked with a "-v" option, a file called "y.output" is produced, with a human-readable description of the parser. The "y.output" corresponding to the above grammar is shown in Fig 2.3.

state 0
\$accept : program \$end
error shift 4
FFCGRAM shift 3
error
program goto 1
program goto 2

state 1
\$accept : program \$end
\$end accept
error
state 2

program : prg$_$$l_ therestofprg

$$_1 : _ (1)

. reduce 1

$$_1 _ goto 5

state 3

prg : PROGRAM_ (2)

. reduce 2

state 4

prg . error_ '4'

. reduce 4

state 5

program : pro$_$1_ therestofprg

error shift 10
DECLAR shift 9

. error

therestofprg _ goto 6
declaration _ goto 7
dec _ goto 8

state 6

program : pro$_$1_ therestofprg_ (2)

. reduce 2

state 7

therestofprg : declaration_content

error shift 14
START shift 13

. error

content _ econtent 11
econtent _ goto 12

state 8

declaration : dec_idenlist CCION $$_6_ type $$_7_ ES

error shift 17
IDENTIFIER shift 16
  . error
idenlist goto 15

state 9
  dec : DECIAFE_   (9)
  . reduce 9

state 10
  dec : error_ES
ES shift 18
  . error

state 11
 therestofproc_ : declaration content_   (5)
  . reduce 5
...

Fig 2.3 Example of Yacc (in human readable form),
generated by invoking flag `-v`.

Yacc invokes two disambiguating rules by default:

(1) In a shift/reduce conflict, the default is to do the shift.

(2) In a reduce/reduce conflict, the default is to reduce by the earlier grammar rule in the input sequence.

2.2.3 Error Handling in Yacc

The token name "error" is reserved for error handling. It suggests places where errors are expected, and recovery might take place. The parser pops its stack until it enters
a state where the token error" is legal. It then behaves as if the token error" were the current lookahead token, and performs the action encountered. The lookahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

Like lex, yacc is written in a portable dialect of C. The action and output subroutine are also in C. Moreover, many of the syntactic conventions of yacc follow C.

2.2.4 lex with yacc

lex is designed to simplify interfacing with yacc. What lex generates is a program named "yylex()", the name required by yacc for its analyzer. Normally, the default main program on the lex library calls this routine, but if yacc is loaded, and its main program is used, yacc will call "yylex()". In this case each lex rule should end with

```
return(token);
```

where the appropriate token value is returned. An easy way to get access to yacc's names for tokens is to compile the lex output file as part of the yacc output file by placing the line

```
#include 'lex.yy.c'
```
in the last section of Yacc input. This causes the scanner to be invoked as a function at the end of the parser. Suppose the grammar is to be named "core.y" and the lexical rules are to named core.l the "NIX command sequence can just be:

    yacc core.y
    lex core.l
    cc y.tab.c -11

Two flags can be invoked to help debugging when compiling 'y.tab.c'.

- **-v**  A file "y.output" is generated, contains a description of the parsing tables and a report on conflicts generated by ambiguities in the grammar.
- **-d**  A file "y.tab.h" is generated with the define statements that associate the Yacc-assigned "token codes" with the user-declared "token names". This allows source files other then 'y.tab.c' to access the token codes.
2.3 INTERMEDIATE CODE GENERATION

Intermediate code generation is an extension of context free grammar. The framework, called a syntax-directed translation scheme, allows subroutines or semantic actions to be attached to the productions of a context-free grammar. The semantic routines generate intermediate code when called at appropriate times by a parser for that grammar.

Four kinds of intermediate code often used in compilers are postfix rotation, syntax trees, quadruples and triples. Fig 2.4 compares each type by showing how the expression:

if a then if c < 0 then a + c else a * c else a + t

A representation of three-address statements known as quadruples was used to generate intermediate codes for each Mini-language. There are four fields in this structure, Cρ, ARG1, ARG2, and RSI (result). The Cρ field contains an internal (integer) code that represents the opcode for example, 272 for "ASSIGN" which is an assignment (:=) operator. For easier reading, this field was translated into a character string every time it was printed out. The ARG1, ARG2, and RSI fields are either programmer-defined names, constants or compiler-generated temporary names. A C "structure" statement was used to contain all this information in each compiler. All quadruples for each Mini-language in this
implementation are stored in an array of pointers to C structures.

<table>
<thead>
<tr>
<th>TYPE</th>
<th>ECRM</th>
<th>NOTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Postfix Fcm</td>
<td>acd-ac+ac*?ac+?</td>
<td>Using ? as a ternary postfix form</td>
</tr>
</tbody>
</table>

**Syntax Tree**

```
if-then-else
/    |
|    |
/ |
if-then-else
/ |
| |
/ + + +
/  \\
\ a c c c a t
```

**Quadruple**

<table>
<thead>
<tr>
<th>Op</th>
<th>Arg1</th>
<th>Arg2</th>
<th>RESULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>a</td>
<td></td>
<td>T1</td>
</tr>
<tr>
<td>'1' if</td>
<td>T1</td>
<td>2</td>
<td>E</td>
</tr>
<tr>
<td>(2)</td>
<td>-</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>(3) if</td>
<td>T2</td>
<td>c</td>
<td>E</td>
</tr>
<tr>
<td>(4) +</td>
<td>a</td>
<td>c</td>
<td>T3</td>
</tr>
<tr>
<td>(5) goto</td>
<td>ε</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(6) *</td>
<td>a</td>
<td>c</td>
<td>T4</td>
</tr>
<tr>
<td>(7) goto</td>
<td>ε</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(8) +</td>
<td>a</td>
<td>t</td>
<td>T5</td>
</tr>
<tr>
<td>(9) goto</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(10)</td>
<td></td>
<td></td>
<td>... next statement ...</td>
</tr>
</tbody>
</table>

**Triple**

<table>
<thead>
<tr>
<th>Op</th>
<th>Arg1</th>
<th>Arg2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>(1) if</td>
<td>(2)</td>
<td>3</td>
</tr>
<tr>
<td>(2) goto</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>(3) -</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>(4) if</td>
<td>(3)</td>
<td>6</td>
</tr>
<tr>
<td>(5) goto</td>
<td>ε</td>
<td></td>
</tr>
<tr>
<td>(6) +</td>
<td>ε</td>
<td>c</td>
</tr>
<tr>
<td>(7) goto</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>(8) *</td>
<td>a</td>
<td>c</td>
</tr>
<tr>
<td>(9) goto</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>(10) +</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>(11) goto</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>(12)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig 2.4 Comparison of each type of intermediate code
A statement such as \( A := I + 10 \) was generated as

\[
\begin{align*}
T1 &= 10 \\
T2 &= I + T1 \\
A &= T2
\end{align*}
\]

Temporary names "\( T \)" followed by a counting number are generated every time an "identifier" such as an integer or string is encountered (\( T1 = 10 \)). The example above can be represented by a set of quadruples as shown in fig 2.5.

<table>
<thead>
<tr>
<th>C1</th>
<th>Arg1</th>
<th>Arg2</th>
<th>ksl</th>
</tr>
</thead>
<tbody>
<tr>
<td>(I)</td>
<td>10</td>
<td></td>
<td>T1</td>
</tr>
<tr>
<td>(1)</td>
<td>I</td>
<td>T1</td>
<td>T2</td>
</tr>
<tr>
<td>(2)</td>
<td>T2</td>
<td></td>
<td>A</td>
</tr>
</tbody>
</table>

Fig 2.5 Quadruple representation of three-address statements.

2.3.1 Syntax-Directed Translation

As mentioned above, three-address code in a quadruple form is used as the intermediate code to be generated. In this section some translations of simple statements based on a part of the grammar in Mini-language Core will be considered.

\[
\begin{align*}
A &\rightarrow \text{id := F} \\
E &\rightarrow [F+]F : [\overline{F}]-F \\
F &\rightarrow [\overline{F}*]C \\
C &\rightarrow \text{id | int | (E)}
\end{align*}
\]

Fig 2.6 An Example of grammars rules in Mini-language Core
Using an example is probably the best way to describe how the translation works. The trace of syntax-directed translation

\[ A := E \times (C + D) \]

is shown in fig 2.7.

<table>
<thead>
<tr>
<th>INPUT</th>
<th>STACK</th>
<th>PIACE</th>
<th>GENERATED CCEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ A := E \times (C + D) ]</td>
<td>[ \text{id} := \text{id} ]</td>
<td>[ A \text{E} ]</td>
<td>[ T1 = F ]</td>
</tr>
<tr>
<td>[ E \times (C + D) ]</td>
<td>[ \text{id} := \text{id} ]</td>
<td>[ A \text{E} ]</td>
<td>[ T1 = F ]</td>
</tr>
<tr>
<td>[ (C + D) ]</td>
<td>[ \text{id} := \text{id} ]</td>
<td>[ A \text{T1} ]</td>
<td>[ T1 = F ]</td>
</tr>
<tr>
<td>[ C + D ]</td>
<td>[ \text{id} := \text{id} ]</td>
<td>[ A \text{T1} ]</td>
<td>[ T1 = F ]</td>
</tr>
<tr>
<td>[ +D ]</td>
<td>[ \text{id} := \text{id} ]</td>
<td>[ A \text{T1} ]</td>
<td>[ T1 = F ]</td>
</tr>
<tr>
<td>[ +D ]</td>
<td>[ \text{id} := \text{id} ]</td>
<td>[ A \text{T1} ]</td>
<td>[ T1 = F ]</td>
</tr>
<tr>
<td>[ +D ]</td>
<td>[ \text{id} := \text{id} ]</td>
<td>[ A \text{T1} ]</td>
<td>[ T1 = F ]</td>
</tr>
<tr>
<td>[ +D ]</td>
<td>[ \text{id} := \text{id} ]</td>
<td>[ A \text{T1} ]</td>
<td>[ T1 = F ]</td>
</tr>
<tr>
<td>[ +D ]</td>
<td>[ \text{id} := \text{id} ]</td>
<td>[ A \text{T1} ]</td>
<td>[ T1 = F ]</td>
</tr>
<tr>
<td>[ +D ]</td>
<td>[ \text{id} := \text{id} ]</td>
<td>[ A \text{T1} ]</td>
<td>[ T1 = F ]</td>
</tr>
</tbody>
</table>

Fig 2.7 Trace of syntax-directed translation.

This example uses a bottom-up parser making the proper shift-reduce decisions to reflect the usual associativity and precedence of operators to operate on \[ A := E \times (C + D) \]. The field PIACE is carried along with grammar symbols in the stack but shown on a stack of its own. Generated quadruples...
are shown with the step just before they are generated.

2.3.2 Control-Flow Representation of Boolean Expressions

Refer to the example at the beginning of this chapter,

if a then if c-d ther a c else a*c else a*t

The a and c-d are supposed to be boolean expressions. Both of them will be translated into a sequence of quadruples. Fig 2.6 gives a general idea of translation for conditional statements such as

if E then S1 else S2

and

while i do S

(a) if-statement
b) while-statement

Fig 2.8 Form of code for constructs using Boolean expressions.

Bottom-up parsing was used and this causes a problem. The critical point in tracing syntax-direct translation for control flow statements is that the actual quadruples, to which the jumps are to be made at the time the jump statements are generated, may not have been generated yet. The code that is generated, therefore, is a series of tracing statements with the targets of the jumps temporarily left unspecified. Each such quadruple will be on one or another list of quadruples to be filled in where the proper location is determined. The subsequent filling in of quadruples is called "backpatching".

Throughout this thesis work, the "backpatching" technique has been used to create a structure (quadruples) for control flow statements. Stacks are used to hold the
addresses of those quadruples which need to be filled later. Each type of control flow statement has its own stack in order to prevent confusion in popping and pushing the particular stack. This also eases the advantage of simpler codes.

2.4 INTERPRETER

Commonly, a programming language is implemented on a computer by a combination of translation and interpretation. A program is first translated from its original form into a form which is more easily executable, and then this executable form of the program is decoded and executed by an interpreter. This was the approach which was used in each Mini-language implementation. For each Mini-language an input program was translated into an easily executed form (quadruples) and then an interpreter was used to interpret each quadruple.

Basically, each input program is translated into quadruples which are saved in an array called aquard'. A function called "xeq" is called if the input program is syntactically correct, to execute each quadruple in 'aquard'. The C programming language was used to code each interpreter. To illustrate the idea of interpretation, fig. 2.6 shows how to perform an interpretation algorithm.
Fig 2.9 Basic procedure for program interpretation and execution.
The "xeq" function does whatever each quadruple tells it to do. After checking the "Cp" field (operator) in each quadruple, a case statement is used to branch to the code for each different kind of quadruple. For example, suppose "i" represents the index of the quadruple array in the 'xeq" function. The following list shows some of the key quadruples in each implemented Mini-language along with the action taken by the interpreter.

<table>
<thead>
<tr>
<th>Cp</th>
<th>Arg1</th>
<th>Arg2</th>
<th>Rsl</th>
<th>&quot;xeq&quot; (interpretation)</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>A</td>
<td>P</td>
<td>T1</td>
<td>T1 = 1 if A &lt; P, T1 = 0 otherwise.</td>
<td>All comparison (&lt; &gt; !=) has the same idea.</td>
</tr>
<tr>
<td>[]</td>
<td>T1</td>
<td>T2</td>
<td>T3</td>
<td>T3 = T1 : T2</td>
<td>Case ITEM in &quot;xeq&quot;. (Mini-language Type</td>
</tr>
<tr>
<td>if</td>
<td>T1</td>
<td>3</td>
<td>7</td>
<td>-if T1 = 1 (true) set i to 3, 7 if it is rct.</td>
<td></td>
</tr>
<tr>
<td>call</td>
<td>0</td>
<td>3</td>
<td>-</td>
<td>-save the index of the next quad in a &quot;calladdr&quot; stack. -transfer control to the called procedure by assigning 0 to i.</td>
<td></td>
</tr>
<tr>
<td>param</td>
<td>A</td>
<td>-</td>
<td>-</td>
<td>-save the value of A by pushing it on the &quot;queu&quot; stack.</td>
<td></td>
</tr>
<tr>
<td>que</td>
<td>W</td>
<td>-</td>
<td>-</td>
<td>-assign W to the value taken from the bottom of &quot;queu&quot; stack.</td>
<td></td>
</tr>
<tr>
<td>param</td>
<td>A</td>
<td>-</td>
<td>-</td>
<td>-save the address of A by pushing it on the &quot;queu&quot; stack.</td>
<td></td>
</tr>
</tbody>
</table>
queaddr W - - assign the location of W to the location taken from the bottom of the "quea" stack.

paraml A - - save the address of A by pushing it on the "quev" stack.

loc w - - assign the location of W to the location taken from the bottom of the "quev" stack.

PROCEDURE 1 - - does nothing.

END 1 - - transfer control back to the calling procedure by setting i to the value popping cut of the calladdr stack.

read A - - get the value of A from the array "in" which reads in the data from the data file.

write A - - print out the value of A in the form 'Rs1 A = ε' if A has the value of ε.

Fig 2.10 List of key quaduples and the result of interpretation
CHAPTER 3

MINI-LANGUAGES

In this chapter, the semantics of each of the implemented Mini-languages will be described briefly. All statements in each language and the symbol table, as well as critical implementation points will be focused on.

3.1 MINI-LANGUAGE CORE

Each Mini-language has the form as shown in Fig 3.1.

```
program
  declare identifier.. ,identifier;
  | declare identifier,...,identifier;
begin
  statement-sequence;
end;
```

Fig 3.1 Format of Mini-language

The BNF (Backus Naur Form) for each is shown in appendix P.

A semicolon is used as a termination symbol for statements. An identifier is alphanumeric and only upper case letters can be used. Reserved words are all in lower case letters. The only identifier type used is integer.

3.1.1 Symbol Table
The symbol table is simply a hash table implemented as an array of pointers to table entries. The algorithm used is a hash search. The incoming name is converted into a small integer, which is then used to index into the array of pointers. An array element points to the beginning of a chain of blocks describing names that have that hash value. A block in a chain is a structure containing pointers to the name, the replacement text, and the next block in the chain. A null next-pointer marks the end of the chain.

Basically, each entry consists of name, type, and value. Type indicates temporary identifiers, identifiers, or reserved words.

3.1.2 Semantics of Mini-language Core

For simplicity, the semantics of Mini-language Core will be described in seven sections as followed:

(1) Declaration

(2) Assignment Statement

(3) Input Statement

(4) Output Statement

(5) If Statement
(6) Loop Statement

(7) Expressions

(8) Comparisons

For each type of statement, "Format" is a structure in the Mini-language. Followed by an example of the statement, "Quadruple format" which bases on the format "Cp Rsl Ar1 Arg2" and any effect in symbol table.

### 2.1.3 Declaration

```
Format: declare identifier, ..., identifier;
        [declare identifier, ..., identifier;]
```

Example: declare A, I, C;

Quadruple format: none

Symbol Table: All identifiers are initialized to 0.

A declaration in Mini-language Core specifies one or more identifiers that can be used as variables in a program. Only positive integers are legal values for each variable.

The range of an integer is between 0 and 99999999. There are only three possible outcomes to the execution of a program.

1) Normal termination. A program terminates normally after the execution of its
last statement.

(2) Abnormal termination. An attempt to execute a meaningless statement which then causes the program to terminate abnormally.

(3) Nontermination.

The exact meaning of a program is defined only for programs that terminate normally.

3.1.4 Assignment Statement

Format: identifier := expression;

Example: A := 10; -- value of A is set to 12

Quadruple Format:
ASSIGN(source variable, target variable, (constant , target variable).
Example: = A 10

Symbol Table: Set the value of "target variable" to the value of the "source variable", or to the "constant".

An assignment statement causes the value of the expression at the time of execution to be associated with a variable. Execution of an assignment statement takes place as follows:
(1) The expression given on the right of the assignment statement is evaluated according to the rules given under expression below. If the expression contains any variables, their current value is used in the evaluation.

(2) The value obtained from the evaluation of the expression becomes the current value of the variable on the left of the assignment.

3.1.5 Input Statement

Format: input identifier [, identifier] .. ;

Example: input A, B, C;

Quadruple Format: RFAD(identifier)
                  [RFAD(identifier)]

Symbol Table: Set the value of 'identifier' to the saved value in the 'in' array. Increase the index of 'in' array.

An input statement causes one or more integer values to be read from an input file named 'data'. Only one value will be assigned to each identifier in the list of identifiers. Each input value must be separated by one or more blank characters, and end-of-line boundaries are treated as single blank characters. For example,

<table>
<thead>
<tr>
<th>&quot;data&quot; input</th>
<th>input statement</th>
<th>result</th>
</tr>
</thead>
</table>
Three kinds of errors that can occur during the execution of an input statement are:

1. **Insufficient data error:** The "data" file contains fewer values than there are identified in the input statement.

2. **Size error:** The integer value read from 'data' file is out of range (≤ 0 or ≥ 99,999.999).

3. **Illegal character error:** One of the characters read from the "data" file is an illegal character (other than a digit or blank).

If any of those errors occur, the program is abnormally terminated.

3.1.6 Output Statement

```
format: output identifier [....identifier...];
example: output A, E, C;
```
Quadruple Format: \texttt{WRITE(identifier)}

Symbol Table: not affected

Execution of an output statement causes the value of each of the variables in the list to be printed. Each value is preceded by "Rsl" (stands for "Result"), the name of the variable and an symbol. For example,

\begin{verbatim}
Output Statement                  Action
output A, B, C;                  Rsl A = 0
                                 Rsl F = 5
                                 Rsl C = 16
\end{verbatim}

The output starts on a new line.

2.1.7 If Statement

\begin{verbatim}
Format: \texttt{if comparison then statement-sequence} \
| \texttt{else statement-sequence} \end{verbatim}

Example: if \texttt{A < F} then
\begin{verbatim}
A := A ; F; 
F := F - A; 
else A := A - F; 
end if;
\end{verbatim}

Quadruple Format:
\begin{verbatim}
IF 'compare result, true target, false target)
\end{verbatim}

Since this is a one pass compiler, the if statement is a critical point in the implementation. A backpatching technique was used to solve the problem generated by the 'for-
ward go to required for the false target". In this section some further ideas about backpatching to generate quadruples will be described.

The idea of backpatching is quite simple. For example, the code

```
PACKPATCH(p,1)
```

means "make each of the quadruples or the list pointed to by p take i as a target'. In other words, keep a list of those addresses that need to be filled later and then fill them with a location at the proper time. The proper time is when some termination like "end if" or "else" is found in an input stream.

It is not too difficult to use backpatching in Yacc, since the Yacc format is in grammar form. The critical point is that the user has to realize that Yacc uses bottom-up parsing which requires that all situations are considered before backpatching a particular statement. For example, consider the nested-if statement shown in Fig 3.2
if \( (A < E) \) then
\[ C := A + B; \]
if \( (E < C) \) then
\[ C := E; \]
else
\[ C := A; \]
end if;
else
if \( (A < F) \) then
\[ A := C; \]
end if;
\[ C := A + B; \]
end if;

Fig 3.2 A nested-if in Mini-language Core.

Fig 3.3 shows how backpatching was used for the above example.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Quadruple Generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>if ( (A &lt; F) ) then</td>
<td></td>
</tr>
<tr>
<td>C := A + F;</td>
<td>100: (&lt;) \ A \ B \</td>
</tr>
<tr>
<td></td>
<td>101: if \ T1 \ 102 \</td>
</tr>
<tr>
<td>if ( (E &lt; C) ) then</td>
<td></td>
</tr>
<tr>
<td>C := E;</td>
<td>102: \ A \ B \</td>
</tr>
<tr>
<td></td>
<td>103: = \ T2 \ C \</td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>C := A;</td>
<td>104: \ (&lt;) \ E \ C \</td>
</tr>
<tr>
<td>end if;</td>
<td>105: if \ T2 \ 106 \</td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td></td>
<td>106: = \ E \ C \</td>
</tr>
<tr>
<td>if ( (A = E) ) then</td>
<td></td>
</tr>
<tr>
<td></td>
<td>107: goto \</td>
</tr>
<tr>
<td></td>
<td>Fix the Rsl field in 106 with 108.</td>
</tr>
<tr>
<td></td>
<td>108: = \ A \ C \</td>
</tr>
<tr>
<td></td>
<td>109: goto \</td>
</tr>
<tr>
<td></td>
<td>Fix the Rsl field in 101 with 110.</td>
</tr>
<tr>
<td></td>
<td>110: = \ A \ E \</td>
</tr>
<tr>
<td></td>
<td>111: if \ T4 \ 112 \</td>
</tr>
</tbody>
</table>
\[ A := C; \]
\[ C := A + E; \]

```plaintext
112: = C A
end if;

113: goto 114
Fix the Rsl field in 111 with 114.

114: - A B C
end if;

115: goto 116
Fix the Arg1 field in 127 and the Arg1 field in 106 with 116.
```

**Fig 3.3** Backpatching corresponding to the nesting if in fig 3.2

Finally, the following intermediate code would be generated in Mini-language Core.

<table>
<thead>
<tr>
<th>Op</th>
<th>Arg1</th>
<th>Arg2</th>
<th>Rsl</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>&lt;</td>
<td>A</td>
<td>P</td>
</tr>
<tr>
<td>101</td>
<td>if</td>
<td>T1</td>
<td>102</td>
</tr>
<tr>
<td>102</td>
<td>-</td>
<td>A</td>
<td>P</td>
</tr>
<tr>
<td>103</td>
<td>=</td>
<td>T2</td>
<td>C</td>
</tr>
<tr>
<td>104</td>
<td>&lt;</td>
<td>P</td>
<td>C</td>
</tr>
<tr>
<td>105</td>
<td>if</td>
<td>T3</td>
<td>106</td>
</tr>
<tr>
<td>106</td>
<td>=</td>
<td>E</td>
<td>C</td>
</tr>
<tr>
<td>107</td>
<td>goto</td>
<td>116</td>
<td></td>
</tr>
<tr>
<td>108</td>
<td>=</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>109</td>
<td>goto</td>
<td>116</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>=</td>
<td>A</td>
<td>T4</td>
</tr>
<tr>
<td>111</td>
<td>if</td>
<td>T4</td>
<td>112</td>
</tr>
<tr>
<td>112</td>
<td>=</td>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>113</td>
<td>goto</td>
<td>114</td>
<td></td>
</tr>
<tr>
<td>114</td>
<td>+</td>
<td>A</td>
<td>P</td>
</tr>
<tr>
<td>115</td>
<td>goto</td>
<td>116</td>
<td></td>
</tr>
<tr>
<td>116</td>
<td>:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>117</td>
<td>:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fig 3.4** Intermediate code for Mini-language Core.
As mentioned, the quadruple format for the if statement is

\[
\text{IF(\text{compare result, true target, false target})}
\]

The "true target" is always the next generated quadruple, "false target" is assigned after a proper 'else' or 'end if' (if there is no else) is encountered.

It is necessary to have some redundant "goto"s; such as quadruple 113 so that the general codes for all situations such as "else" after 'end if" can be generated. This redundancy can be eliminated in the next phase, code optimization, but unfortunately, for this project, an interpreter was implemented instead of using code optimization and code generation. This idea could be considered, for further implementation.

3.1.8 Loop Statement

\[
\text{Format: while comparison loop} \\
:\text{----} \text{---} \text{-----} \text{---} \text{-----} \text{---} \\
\text{statement-sequence} \\
\text{end loop;} \\
\text{---------}
\]

\[
\text{Example: while (WEIGHT \text{\_} 150) loop} \\
\text{ICSS := IAST ICSS \_ 5;} \\
\text{LAST LOSS := ICSS;} \\
\text{WEIGHT := WEIGHT \_ ICSS;} \\
\text{end loop;}
\]

\[
\text{Quadruple Format:} \\
\text{IF(\text{compare result, true target, false target})}
\]
Symbol table: Not affected.

A loop statement is a compound statement that specifies that the statements within the loop are to be executed repeatedly for as long as the comparison at the head of the loop is true.

Actually the loop statement is a version of the if statement. The intermediate code basically used the if and "goto" statements. The following is the intermediate code generated for the above example.

\[
\begin{align*}
\text{IF} &: : \\
\text{CC} &: : \\
100 &: \_tT_\text{IGHT} 150 \_T_1 \\
101 &: \text{if } T1 102 107 \\
102 &: = T2 \\
103 &: + \text{I}A\text{ST}_{-}\text{IOSS} T2 \text{IOSS} \\
104 &: = \text{IOSS} \\
105 &: + \_WIGHT \text{I}A\text{ST}_{-}\text{IOSS} \\
106 &: \text{goto } 100 \\
107 &: : \\
108 &: : \\
\text{IF} 3.5 \text{ Intermediate code for while statement.}
\end{align*}
\]

2.1.9 Comparisons

Format: (operand < operand) or (operand = operand) or (operand > operand) or (operand != operand)

Example: (SMALL < Fig)

Quadruple Format:

\[
\begin{align*}
\text{LT GT} \\
\text{EQ NEQ(first value,second value,target value)}
\end{align*}
\]
Symbol Table: "Target variable" (a temporary name preceded by 'T' followed immediately by counting number) is created in a symbol table with a boolean type. The value is set to the truth value ("true" (1), or "false" (0)) of the particular relationship between the first and the second values.

A comparison consists of two operands separated by one of the comparison operators, <, >, <=, !=. Should the evaluation of one of the operands lead to an error, the program will be terminated abnormally.

3.1.12 Expression

Format: operand - operand or operand - operand or operand * operand

Example: A + F * C

Quadruple Format:
ADD MUI(first value, second value, target variable)
MINUS

Symbol Table: The value of the "target variable" is set to the result of applying the particular function between the "first value" and "second value".

The operators are evaluated in order of decreasing precedence, defined by the rules:

(1) The operator * has higher precedence than the + and operators, which have equal precedence.
(2) Operators of equal precedence are evaluated in textual order from left to right.

(3) An expression enclosed in parenthesis is evaluated to a single value before other operators.

Two errors can arise during the evaluation of an expression.

(1) Undefined value error. A variable in the expression has not previously had a value assigned to it.

(2) Overflow error. One of the operations leads to a value greater than the maximum permitted value defined by the implementation.

The occurrence of either of these errors causes abnormal termination of the program.

In addition to the rules given for the construction of a program in Mini-language Core, there are two constraints;

(1) All identifiers used in the statements of the program must be declared.

(2) No identifier may appear more than once.

A comment is introduced by two contiguous hyphen symbols (that is, --). These two symbols and the remaining characters on the same line are treated as the text of the
comment and have no effect on program execution. A comment can occur in the program at any point where a blank may appear.

As mentioned above lower case letters are used for keywords and upper case letters for identifiers. This serves the dual purpose of differentiating one kind of statement from another and of making more readable programs.

All keywords are reserved so there is no danger of a loss of readability due to the programs choosing identifiers that clash with keywords. This also provides fixed markers in the syntax that allow the compiler to make better recovery in the face of syntax errors and give more meaningful error messages.
3.2 MINI-LANGUAGE D

Mini-language D is used to focus on control statements, such as loop and if statements. In mini-language D, there are three ways in which the sequence of statement execution may be specified:

1. Sequential execution. The statements are executed precisely in the order in which they are written.

2. Conditional or selective execution. Expressed by the if statement.

3. Iterative execution. For example, loop statement.

If the condition expression has the value false initially, the body of the loop is never executed; and the loop statement has no net effect.

A condition-expression is either a single condition or a pair of conditions separated by one of the logical operators "and" and "or". Program in Mini-language D, of course, have variables and all variables in a program must be declared. Only integer type is used.

Since almost all statements in Mini-language D had been described previously in Mini-language Core, the following
will focus on the main idea of D structures

**D Structure**

A D structure, D for Dijkstra (as in Bruno and Steiglitz 1972), is a class of simple control structures. A D structure is either a

* Basic action: For example, an assignment statement, procedure call, or the input-output statement; or it is constructed from simple D-structures, each using one of the following forms:

* Sequence
  
  $s_1 \ s_2 \ ... \ s_n$

  of two or more D-structures $s_1$ through $s_n$.

* Conditional structure

  
  \[
  \text{if } c \ \text{then} \ 
  s_1
  \]

  
  \[
  \text{else} \ 
  s_2
  \]

  \[
  \text{end if;}
  \]

  where $c$ is a condition and $s_1$ and $s_2$ are D-structures.

* Iterative structures

  
  \[
  \text{while } c \ \text{loop}
  \]

  
  \[
  s
  \]

  \[
  \text{end loop;}
  \]

  where $c$ is a condition and $s$ is a D-structure.

Fig 3.6 shows a Mini-language D program that the true branch is always shown on the left of the node.
This program reads in an integer value representing the time on a 24-hour clock and prints out the corresponding 12-hour clock time. If the input value does not represent a correct time, the input value is printed.

```plaintext
declare TIME, HOURS_AND_MINUTES, HOURS, MINUTES, AM_or_PM;
begin
    input TIME;
    HOURS_AND_MINUTES := TIME;
    HOURS := 0;
    while (HOURS_AND_MINUTES > 100) loop
        HOURS_AND_MINUTES := HOURS_AND_MINUTES - 100;
        HOURS := HOURS - 1;
    end loop;
    MINUTES := HOURS_AND_MINUTES;

    if (HOURS > 23) then
        if (HOURS = 24) and (MINUTES = 0) then
            AM_or_PM := 0;
            HOURS := 12;
            output HOURS, MINUTES, AM_or_PM;
        else
            output TIME;
        end if;
    else
        if (MINUTES > 59) then
            output TIME;
        else
            AM_or_PM := 0;
            if (HOURS = 0) then
                HOURS := 12;
            else
                if (HOURS > 11) then
                    AM_or_PM := 1;
                    if (HOURS > 12) then
                        HOURS := HOURS - 12;
                    end if;
                end if;
            end if;
            output HOURS, MINUTES, AM_or_PM;
        end if;
    end if;
end;
```

Fig. 3.6 A Mini-language D program
The prevailing winds seem to be that using the goto statement in any programs is not a good programming style. Currently, most of programming languages ever Ada, still have the goto statement. Since, Ledgard wrote Mini-language D to emphasize good programming structure, he ignored the goto statement and enforced the idea of one-in, one-out structures.

The basic actions of one-in, one-out structure are such that no transfer of control can occur during their execution. That is, control enters by only one path and leaves by only one path. D-structure is built from one-in, one-out structures. A program that is constructed entirely from D-structure is itself a D-structure. Consequently, it will have only one entry and one exit. The control schemes of Mini-language D correspond exactly to the construction rules for D-structures and as a result, all programs written in Mini-language D are D-structured. To make this clear, if 3.7 shows a program that is not D-structure (uses goto statement).
I1:  a1;
I2:  a2;
I3:  a3;
    if C1 then
         a4;
         goto I2;
    end if;

    if C2 then
         a5;
         goto I4;
    end if;

    a6;
    if C3 then
         a7;
         goto I4;
    else
         a8;
         goto I3;
    end if;
I4:  a9;
    if C4 then
         goto I1;
    end if;

Fig 3.7 An Example of a program that is not a D-structure.

Furthermore, Iedgard expressed the idea of the study of the classic theorem of Poehm and Jacopin [1966] in D-structures. He simply explained the proof of the theorem (18). The basic conclusion of that theorem can be stated simply as:

For any proper program there exist an equivalent program that is a D-structured.
By "any proper program" he means any computer program, no matter what control structures are used, provided:

1. There is precisely one entry and one exit to the program.

2. For every node in the flowgraph representation of the program, there is at least one path from the entry point, through that node, to the exit point. The latter restriction rules out programs containing infinite loops and statements that are not reached by the flow of control from the programs at entry point.

By "equivalent program", he means a program that will always give the same result as the original one for the same input data. Two equivalent programs may have very different flowgraphs. For example, compare two programs that calculate the square root of their input. One obtains the result by successive approximation, while the other uses a table look-up method. These two programs will be equivalent if their results are exactly equal for all possible input values.

The impact of the theorem is that it is possible to write any program as a D-structure. The theorem guarantees that any problem can be written using only D-structures. In particular, if a programming language includes only the following control statements:
(1) Sequences of one or more statements.

(2) Conditional Statements of the form

if condition then
   statements ..
else
   statements ..
end if;

(3) Loops of the form

while condition loop
   statements ...
end loop;

or their equivalent, then this is all needed, at least theoretically.
3.3 MINI-LANGUAGE TYPE

The Mini-language Type serves as a basis for a discussion of the concept of variable types in programming languages. This discussion will be limited to the primitive types of a language.

As usual, a program in Mini-language Type consists of a sequence of declarations followed by a sequence of statements. The declarations specify the type of value that is to be associated with each identifier. The statements define the operations to be performed on values associated with declared variables.

The types in Mini-language Type are either simple or composite. The simple types include integers, strings of characters (for example, 'TAD' and '4RK'), and the boolean values 'true' and 'false'. The composite types in Mini-language Type are arrays of a given simple type and record structures. All identifiers referred to in the program must be declared exactly once.

There are four varieties of statement in Mini-language Type, each of the usual form:

1. An assignment statement: Both the variable and the expression must be of the same simple type.
(2) An if statement: The conditional expression must be of boolean type.

(3) An input statement.

(4) An output statement.

Variables may be combined by operators in an expression to form new values. The operators +, -, and * are defined over integers to yield their conventional result.

The relational operations > and < are defined over integers and give a result of type boolean. The equality operators = and != are defined over any two objects of the same simple type and also yield a result of type boolean.

The operators 'and' and 'or' are defined over two boolean values and perform the boolean "and" and "or" operations on the two values. The operator 'cat' is defined over two string values and yields the string consisting of the concatenation of the two values.

3.3.1 Symbol Table Management

The symbol table Management in Mini-language Type is quite different from that in Mini-language Core and Mini-language E. There are four tables; Identifier table, Variable table, Type table and Store table.
(1) Identifier table: 'idt tab' has 2 fields 'name' and 'varaddr'.

'name' identifies each variable or identifier name.

'varaddr' is a pointer to the Variable table.

The table is indexed by an identifier number issued by analyzer (1, 2, 3, ..., IDMAX).

(2) Variable table: 'vartab' has 2 fields 'typeaddr' and 'storeaddr'.

'typeaddr' contains a pointer to a Type table row for this variable's declared type.

'storeaddr' contains a pointer to Store table.

(3) Store table: 'storetab' is just an array contain value of each variable and identifier.

(4) Type table: 'tupetab' with four fields included.

'Sort'

'field1'
The 'Sort' field will contain a number representing a type. The following explains all possible types used in Mini-language Type and how each type is represented in the Type table.

```
REC : record

RECORD  size  chain  0

size : The number of fields of each record.

chain : A pointer to Type table row where the first field may be found.

RECFLD : field (of a record)

RECFLD  id  chain  type

id : A symbol table number (after using hash function) of field designator 'id'.
chair : A pointer to Type table row of next field in record; zero if last.

type : A pointer to Type table row of type of this field.

ARRAY : array

<table>
<thead>
<tr>
<th>ARRAY</th>
<th>size</th>
<th>chain</th>
<th>type</th>
</tr>
</thead>
</table>

size : The total size of an array of this type in storage ('storetab').

chair : A pointer to Type table row of first subscript range.

type : type of elements (Type row number).

SSRANGE : subscript range

<table>
<thead>
<tr>
<th>SSRANGE</th>
<th>str</th>
<th>chain</th>
<th>type</th>
</tr>
</thead>
</table>

str : The stride of this range - net change in address for unit change in this
subscript (assuming element size one).

chain: A pointer to the Type table row of
next subscript range, zero if last.

type: A pointer to the Type table row of
subscript values; must be of integer
 subrange, string subrange, boolean
 subrange.

INTSUP : integer subrange
---------
STRSUP : string subrange
---------
ECSUP  : boolean subrange
---------

INTSUP  field1  lb   ub
STRSUP  
ECSUP   

field1: For array, this field contains constant
 number identify type integer if INTSUP.
For record structure, this field
contains counting number of record
field.

lb: Indicates the lower bound of the array.
ub: Upper bound of each range.

3.3.2 More about Composite Type

Before further consideration, one should realize that Mini-language Type eliminates records of arrays and vice versa.

In Mini-language Core the translation of assignment statements having only simply names as operands. In Mini-language Type Ledgard introduces array references and record structures as operands. For array reference such as $A[i][j]$, the approach used here is to produce three-address codes (quadruple) that computes the offset of $A[i][j]$ from the base of array $A$ and then performs an indexing operation.

The array statement in Mini-language Type has unlimited positive bounds. Any positive number can be assigned to either lower bound or upper bound which, of course, includes zero. The main idea of row major form is to store the array in a block in such a fashion that if one scans the block from top to bottom, and notes the indices of the word stored in each location, the rightmost index varies fastest, the second rightmost varies next fastest and the leftmost index varies slowest.

Let's consider a k-dimensional array $A[i_1, i_2, \ldots, i_k]$ where $i_th$ index runs from lower bound $i$ ($\leq i$) up to some di
for each dimension $i=1,2,\ldots,k$. Mathematically, we give this element $A[i_1,i_2,\ldots,i_k]$ offset

$$(i_1-LE_1)D_2D_3\ldots D_k + (i_2-LE_2)D_3D_4\ldots D_k + \ldots + (i(k-1)-LE(k-1))D_k + (i_k-1P_k)$$

from the first word of the array. We can rewrite (T.1) as

$$\sum_{j=1}^{k} (i_j-LB_j)D_j \rightarrow D_j = \sum_{k=j+1}^{k} D_k \quad \text{and} \quad D_k = 1.$$ 

$D_j$ can be calculated in advance in the lexical analyzer phase. Finally, the three-address code for $i$, for example, $A[i_1,i_2,i_3,i_4]$ is

- $T1 = i_1 - LE_1$
- $T2 = T2 \times D1$
- $T3 = i_2 - LE_2$
- $T4 = T3 \times D2$
- $T5 = T2 + T4$
- $T6 = i_3 - LE_3$
- $T7 = T6 \times D2$
- $T8 = T5 + T7$
- $T9 = i_4 - LE_4$
- $T10 = T9 \times D3$
- $T11 = T8 + T10$
- $T12 = \text{addr}(A)$
- $T13 [\] T11 T12$

Fig 3.8 Example of quadruple for array type in Mini-language Type.

The '[' has the meaning of addition when the opcode appears in the interpreter.
3.3.3 Record Type

Basically, a record type contains a collection of components, each of which may be of a different type. Each component has a name and a value. The following is an example of a record structure declaration in Mini-language Type for the record of a person's drivers license.

```
declare LICENSE :
  record
    DRIVER : record
      FIRST_NAME : string;
      MIDDLE_NAME : string;
      LAST_NAME : string;
    end record;
    LICENSE_NUM : string;
    EXPIRATION_DATE : record
      MONTH : integer;
      DAY : integer;
      YEAR : integer;
    end record;
    DRIVING_CODE : string;
```

Fig 3.9 Example of a record structure declaration of record type in Mini-language Type

The translation approach used here is to leave a reference such as LICENSE.FIRST_NAME intact in the three address code. The interpreter will take care of finding the actual address of each reference. This task is assigned to a file called 'get.c' which contains two main procedures 'get' and 'getplace'. What 'get' procedure does, basically, is to look at the name 'LICENSE', get an index from the 'varaddr' field in 'id' table. Using this index to indicate the row to look at in 'var' table. Get the 'type' index from the
`typeaddr` in `var` table and at the same time save the number(A) in `storeaddr` field which indicates the row in `store` table where the value of `ICENSE` structure is started (base address of the structure). At this point the `type` index is used to specify where `ICENSE` record starts in `type` table. `getplace` will be called at this point to start looking down the `type` table by following the chain at each row until the last subrange of each reference, `FIRSTNAME` in this example. is found. The counting number for each subrange which was kept in field1 is saved (B). The counting number is added to the base address of this structure (LICENSE) A + B). The answer will be the index in `store` table pointing to the value, `LICENSE.FIRSTNAME` in this example.
3.4 MINI-LANGUAGE PROCEDURES

Subprograms allow the programmer to package computations and parameterize their behavior. There are two forms of subprograms, procedures and functions. A procedure subprogram is a sequence of actions that is invoked by a call statement. A function subprogram is a sequence of computations that results in a single value and is invoked from within an expression. Usually, control returns to the point of invocation after execution of the subprogram, thus forming another one in, one-out control structure.

The Mini-language Procedures, is used to demonstrate some techniques of passing data between the subprogram and the program that calls it. Only global variables are discussed here. More about scope mechanism is discussed in the Mini-language Scope. Also a special property, recursion, is described in the Mini-language Apply.

A program in Mini-language Procedures, consists of a sequence of declarations followed by a sequence of statements. There are two types of declarations, one for variables and one for procedures.

Variable declarations introduce simple variables and arrays of integer only. Unlike Mini-language Type, array variables contain an unspecified number of components. For this implementation each array has a range between 0 and lower
bound) and 100 (upper bound). For example, one may have:

```plaintext
declare X,Y,GOOD;  -- three integer-valued variables.
declare A,B:array;  -- two arrays with integer components.
```

All variables used in the statement part of a program must be declared exactly once.

A procedure declaration defines a procedure subprogram and contains the following parts.

1. an identifier name for a procedure.
2. the names of parameters and their modes.
3. the declaration of any variables local to the procedure,
4. a sequence of statements comprising the body of the procedure.

All variables used within the body of a procedure must either be declared in the procedure or be parameters.

There are four types of statements in Mini-language Procedures. An assignment statement, an input statement which allows input value to be read in from the 'data' file, an output statement and a call statement. A call statement consists of the name of a declared procedure and arguments corresponding to each parameter associated with the
During execution of a call statement, two things take place:

1. A correspondence between the arguments in the argument list and the parameters in the procedure is established in left-to-right order. The i-th argument corresponds to the i-th parameter. The rules for passing the arguments to the procedure being called are then applied to each separately.

2. Control is transferred to the first executable statement of the body in the invoked procedure.

When the last statement in the called procedure has been executed, control is returned to the statement following the call statement.

The way in which the argument is passed to its corresponding parameter depends on the mode of the parameter. There are five ways of passing parameters in the Mini-Language Procedure as follows:

(1) Pass by Value
(2) Pass by Result
(3) Pass by Value-result
(4) **Pass by Location**

(5) **Pass by Name** 'not included in this implementation'.

3.4.1 **Pass by Value**

The parameter acts as a local variable belonging to the procedure. This local variable is initialized with the value of the corresponding argument. Since the parameter is purely a local variable, any change of its value during execution of the procedure has no effect on the corresponding argument. An argument passed by value must be an integer-valued expression.

3.4.2 **Pass by Result**

The parameters again act as local variables, but their values must be initialized locally within the procedure body. After the statements of the body have been executed, the value of the parameters are assigned to the corresponding arguments. The arguments must be variables.

Arguments passed by value are expressions that provide 'inputs' to a procedure; arguments passed by result are variables that receive 'outputs' from a procedure.

3.4.3 **Pass by Value-result**
Pass by value-result combines the characteristics of pass by value and of pass by result. The parameter is considered a variable local to the procedure: its initial value is given by the value of the corresponding argument, and the final value of the parameter is assigned to the argument on completion of execution of the procedure.

3.4.4 Pass by Location

The parameter is considered a local variable of the procedure, but its location is the location of the argument. Thus, any reference to the value of the parameter is considered a reference to the value of the argument, and any assignment to the parameter is an assignment to the corresponding argument, thus changing the argument's value.

Pass by location (or reference) is quite similar to pass by value-result. The difference is, with pass by location, any change in the value of a parameter is immediately reflected as a change in the value of the corresponding argument. With pass by value-result, the value of the argument changes only on final exit from the called subprogram.

3.4.5 Pass by Name

Pass by Name allows results to be transmitted back from a subprogram through assignments to the corresponding param-
3.4.6 Symbol Table Management

There are four symbol tables in Mini-language Procedures. The main table is called 'idtab'. As the table name suggests, it contains all identifier and variable names used in the program. The table is divided into three fields:

1. **Name**: Contains names of identifiers and variables
2. **Type**: The type of each variable or identifier. Possible types are
   - **FAR** ('parameter')
   - **PROCEDURE** ('name of procedure')
   - **SIMPLE** ('integer in main program')
   - **ARRAY** ('array')
3. **Store**: Pointer to the other three tables, depends on Type field.

<table>
<thead>
<tr>
<th>Possible Type</th>
<th>Point to</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAR</td>
<td>'partat' (parameter table)</td>
</tr>
<tr>
<td>PROCEDURE</td>
<td>'proctab' (procedure table)</td>
</tr>
<tr>
<td>SIMPLE</td>
<td>'storetab' (store table)</td>
</tr>
<tr>
<td>ARRAY</td>
<td>'storetab' (store table)</td>
</tr>
</tbody>
</table>
A unique number is assigned to each procedure name. This unique number is stored in the Store field, its Type is PROCEDURE, and it is used as an index in 'proctab', the procedure table, which has four fields as following:

(1) Unique : Unique number for each procedure.

(2) Guard# : The number of the quadruple where sequence of the corresponding procedure starts.

(3) Count : Number of parameters in the procedure.

(4) Par : A pointer to the first parameter of the procedure in 'partab'

Every time the procedure statement is invoked all parameters, from left to right, are stored in 'partab' from top to bottom. The pointer in Par field in 'proctab' is the number of the first (the left most parameter) in the parameter statement. These numbers are also used as the index in 'partab' which has four fields as following:

(1) Pindex : Unique number for each parameter.

(2) Powner : Unique number representing the name of the procedure which the parameter belongs to
(3) Ftype : The mode of the parameter (Value, Result, Value-result, Location)

(4) Fstore : A pointer to 'storetab'.

The store table, 'storetab', is just a one dimensional array contains values of variables, identifiers and parameters. Only integers are considered in Mini-language Procedures.
CHAPTER 4

CONCLUSIONS

Mini-languages are good examples of programming languages which provide simplicity and compactness. The purpose of Henry Ledgard [The Program Language Landscape] on Mini-languages was to provide an easier way to study programming languages. Each Mini-language focuses on one important feature of programming languages at a time. For example, Mini-language Procedures concentrates only on procedure call statement, and Mini-language Type concentrates on the variable type statement.

The four compilers implemented here for this thesis work are fairly modular, so additions or modifications could be made to them with few problems. Mini-languages could be used as a tool for research into the problems of various languages. Perhaps, these four compilers could be used by the starting programmers to test their ability in understanding each feature of programming languages.

This chapter contains a discussion of what might be changed in this thesis work, or what might be added to it.

4.1 New Features
A couple of the compiler phases that were eliminated from the design of each of the four compilers were the code optimization and the code generation phases. The intent of the code optimizer is to remove redundant or useless instructions that are generated because of the simplicity of the intermediate code generation. For example, a "go to" statement

```
  ...
  100  goto 101
  101  goto 102
  102  
  103  
  104  
```

which might happen in some situations with an "if" statement and a "loop" statement.

Since the purpose of this thesis is to learn more about programming languages, the code generation phase for one particular machine was left out so that the writer had more time to concentrate on other features of the programming languages. Anyway, the interpreter for each Mini-langauge was written to test each compiler. Further implementation can be made from this point. The code optimization phase and the code generation phase should be able to be implemented without any difficulties.
4.2 Suggested Changes

Because of the time factor, the various procedures of this thesis work were kept simple which meant that they were not always efficient. One possible change would be to modify the organization of the arrays currently used in the programs generating each compiler. Such arrays, for example, the 'Temporary' array which creates temporary names 'T1'..."T1000" used in generating intermediate code grow pretty fast. Besides this, stacks are used in each implementation to keep things simple. As with the arrays, stacks used here grow fast. A more efficient stack management technique might be used to improve memory usage efficiency. lex input and yacc input written here are quite simple and easy to understand, this causes, sometimes, quite cumbersome code.

In any case, much effort has been put into this thesis. Hopefully, this thesis will be of some use in the future.


(5) Eastman C.M., Department of Math and Computer Science, Florida State Univ., Tallahassee, Florida, Lexical Characteristics of Keywords in High Level Programming Languages", Compsac 81, IEEE Computer Society's Fifth International Computer Software and Applications


(16) Lesk M.F. and Schmidt F., "Lex - A Lexical Analyzer Generator", Bell Laboratories, Murray Hill, New Jersey.


APPENDICES
6.1 APPENDIX A

In this appendix all three Mini-languages implemented in this thesis work are shown in PNF form.

Context Free Syntax of Mini-language in PNF

Some extensions to PNF were used to describe the grammars of Mini-languages.

(1) Optional Items: There are enclosed in brackets, thus introducing the additional meta-symbols [ and ].

(2) Sequences: The ellipsis symbol ( . . ) is introduced as another meta-symbol to indicate the repetition of the preceding categories contained in brackets an arbitrary number of items.

(3) Typefaces: The names of PNF categories will be written without < and but in a typeface different from that of that language being defined.
As with PNF, where there is a clash between a meta-symbol and a symbol of the Miri-languages, the symbol that is part of the Miri-language will be underlined. For example.

\[
\text{variable} ::= \text{identifier} \\
\quad | \text{identifier [expression]}
\]
Table F.1 Context Free Syntax of Mini-language Core in PNS

```
program ::= program
  <declaration-sequence>
begin
  <statement-sequence>
end;

<declaration-sequence> ::= <declaration>
  | <declaration> <declaration-sequence>

<statement-sequence> ::= <statement>
  | <statement> <statement-sequence>

<declaration> ::= declare <identifier-list>;

<identifier-list> ::= <identifier>
  | <identifier> , <identifier-list>

<statement> ::= <assignment-statement>
  | <if-statement>
  | <loop-statement>
  | <input-statement>
  | <output-statement>

<assignment-statement> ::= <identifier> := <expression>

<if-statement> ::= if <comparison> then
  <statement-sequence>
end if;
  | if <comparison> then
  <statement-sequence>
else
  <statement-sequence>
end if;

<loop-statement> ::= while <comparison> loop
  <statement-sequence>
end loop;

<input-statement> ::= input <identifier-list> ;

<output-statement> ::= output <identifier-list> ;

<comparison> ::= <operand> = <operand>
  | <operand> < operand>
  | <operand> > < operand>
  | <operand> <= < operand>
  | <operand> >= < operand>

<expression> ::= <factor>
```


| <expression> | + | <factor> |<expression> | - | <factor> |
|<factor> | ::= | <operand> |<factor> | * | <operand> |
|<operand> | ::= | <integer> |<identifier> |<expression> |
|<identifier> | ::= | <letter> |<identifier> |<letter> |<identifier> |<letter> |
|<integer> | ::= | <digit> |<integer> |<digit> |
|<letter> | ::= | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
|<digit> | ::= | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
Table 1.2 Context Free Syntax of Mini-language D in PNF

<table>
<thead>
<tr>
<th>Syntax Element</th>
<th>Grammar Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;program&gt;</code></td>
<td><code>::= program</code></td>
</tr>
<tr>
<td></td>
<td><code>   </code>declaration-sequence`</td>
</tr>
<tr>
<td></td>
<td><code>   begin</code></td>
</tr>
<tr>
<td></td>
<td><code>   </code>statement-sequence`</td>
</tr>
<tr>
<td></td>
<td><code>   end;</code></td>
</tr>
<tr>
<td><code>&lt;declaration-sequence&gt;</code></td>
<td><code>::= declaration</code></td>
</tr>
<tr>
<td></td>
<td><code>   declaration-sequence</code>.</td>
</tr>
<tr>
<td><code>&lt;statement-sequence&gt;</code></td>
<td><code>::= statement</code></td>
</tr>
<tr>
<td></td>
<td><code>   statement</code></td>
</tr>
<tr>
<td></td>
<td><code>   statement-sequence</code></td>
</tr>
<tr>
<td><code>&lt;declaration&gt;</code></td>
<td><code>::= declare identifier-list;</code></td>
</tr>
<tr>
<td><code>&lt;identifier-list&gt;</code></td>
<td><code>::= identifier</code></td>
</tr>
<tr>
<td></td>
<td><code>   identifier . identifier-list</code></td>
</tr>
<tr>
<td><code>&lt;statement&gt;</code></td>
<td><code>::= assignment-statement</code></td>
</tr>
<tr>
<td></td>
<td><code>   if-statement</code></td>
</tr>
<tr>
<td></td>
<td><code>   loop-statement</code></td>
</tr>
<tr>
<td></td>
<td><code>   input-statement</code></td>
</tr>
<tr>
<td></td>
<td><code>   output-statement</code></td>
</tr>
<tr>
<td><code>&lt;assignment-statement&gt;</code></td>
<td><code>::= identifier := expression;</code></td>
</tr>
<tr>
<td><code>&lt;if-statement&gt;</code></td>
<td><code>::= if condition-expression then</code></td>
</tr>
<tr>
<td></td>
<td><code>   </code>statement-sequence`</td>
</tr>
<tr>
<td></td>
<td><code>[ else</code></td>
</tr>
<tr>
<td></td>
<td><code>   </code>statement-sequence<code> </code>]`</td>
</tr>
<tr>
<td></td>
<td><code>   end if;</code></td>
</tr>
<tr>
<td><code>&lt;loop-statement&gt;</code></td>
<td><code>::= while condition-expression loop</code></td>
</tr>
<tr>
<td></td>
<td><code>   </code>statement-sequence`</td>
</tr>
<tr>
<td></td>
<td><code>   end loop;</code></td>
</tr>
<tr>
<td><code>&lt;input-statement&gt;</code></td>
<td><code>::= input identifier-list;</code></td>
</tr>
<tr>
<td><code>&lt;output-statement&gt;</code></td>
<td><code>::= output identifier-list;</code></td>
</tr>
<tr>
<td><code>&lt;condition-expression&gt;</code></td>
<td><code>::= [ &lt;condition . and ] &lt;condition</code></td>
</tr>
<tr>
<td></td>
<td><code>   [ &lt;condition . or ] &lt;condition</code></td>
</tr>
<tr>
<td><code>&lt;condition&gt;</code></td>
<td><code>::= comparison</code></td>
</tr>
<tr>
<td></td>
<td><code>   condition-expression</code> <code>)</code></td>
</tr>
<tr>
<td><code>&lt;comparison&gt;</code></td>
<td><code>::= ( operand &lt;comparison-operator&gt; operand</code></td>
</tr>
<tr>
<td><code>&lt;integer-expression&gt;</code></td>
<td><code>::= [ operand + ] operand</code></td>
</tr>
</tbody>
</table>
\[ \text{|} \quad (\text{operand} - \text{operand}) \]

\[ \text{<operand>} \quad ::= \quad \text{<integer>} \quad |
\quad \text{<identifier>} \quad |
\quad (\text{<integer-expression>}) \]

\[ \text{<comparison-operator>} \quad ::= \quad . \quad | \quad = \quad | \quad ! \quad | \quad : \]

\[ \text{<identifier>} \quad ::= \quad \text{<letter>} \quad |
\quad (\text{<identifier>}.\text{<letter>}) \quad |
\quad \text{<identifier>} \quad \text{-} \quad \text{<letter>} \]

\[ \text{<integer>} \quad ::= \quad \text{<digit>} \quad \mid \quad \text{<integer>\text{-<digit>}} \]

\[ \text{<letter>} \quad ::= \quad \text{A} \quad | \quad \text{B} \quad | \quad \text{C} \quad | \quad \text{D} \quad | \quad \text{E} \quad | \quad \text{F} \quad | \quad \text{G} \quad | \quad \text{H} \quad | \quad \text{I} \quad |
\quad \text{J} \quad | \quad \text{K} \quad | \quad \text{L} \quad | \quad \text{M} \quad | \quad \text{N} \quad | \quad \text{O} \quad | \quad \text{P} \quad | \quad \text{Q} \quad | \quad \text{R} \quad |
\quad \text{S} \quad | \quad \text{T} \quad | \quad \text{U} \quad | \quad \text{V} \quad | \quad \text{W} \quad | \quad \text{X} \quad | \quad \text{Y} \quad | \quad \text{Z} \]

\[ \text{<digit>} \quad ::= \quad \text{0} \quad | \quad 1 \quad | \quad 2 \quad | \quad 3 \quad | \quad 4 \quad | \quad 5 \quad | \quad 6 \quad | \quad 7 \quad | \quad 8 \quad | \quad 9 \]
Table E.3 Context Free Syntax of Mini-language Type in PNI

\[
\begin{align*}
\text{<program>} &::= \text{Program} \\
&\quad \langle \text{declaration-sequence} \rangle \\
&\quad \begin{array}{l}
\text{begin} \\
\quad \langle \text{statement-sequence} \rangle \\
\text{end;}
\end{array} \\
\text{<declaration-sequence>} &::= \langle \text{declaration} \rangle \\
&\quad \langle \text{declaration} \rangle \langle \text{declaration-sequence} \rangle \\
\text{<statement-sequence>} &::= \langle \text{statement} \rangle \\
&\quad \langle \text{statement} \rangle \langle \text{statement-sequence} \rangle \\
\text{<declaration>} &::= \text{declare} \quad \langle \text{identifier-list} \rangle : \langle \text{type} \rangle \\
\text{<identifier-list>} &::= \langle \text{identifier} \rangle \\
&\quad \langle \text{identifier} \rangle , \langle \text{identifier-list} \rangle \\
\text{<type>} &::= \langle \text{simple-type} \rangle \\
&\quad \langle \text{array-type} \rangle \\
&\quad \langle \text{record-type} \rangle \\
\text{simple-type} &::= \langle \text{integer} \rangle \\
&\quad \langle \text{string} \rangle \\
&\quad \langle \text{boolean} \rangle \\
\text{array-type} &::= \text{array} \ [ \langle \text{bounds} \rangle ] \ \text{of} \ \langle \text{type} \rangle \\
\text{record-type} &::= \text{record} \\
&\quad \langle \text{identifier} \rangle : \langle \text{type} \rangle ; \\
&\quad \langle \text{identifier} \rangle : \langle \text{type} \rangle ; \ldots \\
&\quad \text{end record;} \\
\text{<bounds>} &::= \langle \text{integer} \rangle \ldots \langle \text{integer} \rangle \\
\text{<statement>} &::= \langle \text{assignment-statement} \rangle \\
&\quad \langle \text{if-statement} \rangle \\
&\quad \langle \text{input-statement} \rangle \\
\text{assignment-statement} &::= \langle \text{variable} \ := \langle \text{expression} \rangle \rangle \\
\text{if-statement} &::= \text{if} \ \langle \text{expression} \rangle \ \text{then} \\
&\quad \langle \text{statement-sequence} \rangle \\
&\quad \langle \text{else} \rangle \\
&\quad \langle \text{statement-sequence} \rangle \\
&\quad \text{end if;} \\
\text{input-statement} &::= \text{input} \ \langle \text{variable} \ [ , \ \langle \text{variable} \rangle \ldots \rangle \\
\end{align*}
\]
output-statement ::= output <variable> [ , <variable> ] ... ;
<expression> ::= | <operand> <operator> | <operand
<operand> ::= <variable>
| <integer>
| <string>
| <boolean>
| [ <expression> ]

<string> ::= ' <character> .. '
<boolean> ::= true
| false
<operator> ::= < | = | != | > | < | - | * | /
| and | or
<character> ::= <letter>
| <digit>
| <special-character
<special-character> ::= (sp) | + | - | * | / | : | ;
| . | . | $ | ? | = | != | > | <
<letter> ::= A | B | C | D | E | F | G | H | I
| J | K | L | M | N | O | P | Q | R
| S | T | U | V | W | X | Y | Z
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Note: (sp) means space or blank character.
Table F.4 Context Free Syntax of Mini-language Procedures in BNE

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>program</td>
<td>:= program &lt;variable-declaration&gt; procedure-declaration begin  &lt;statement-sequence&gt; end;</td>
</tr>
<tr>
<td>variable-declaration</td>
<td>::= &lt;declaration&gt; &lt;declaration&gt; &lt;variable-declaration&gt;</td>
</tr>
<tr>
<td>procedure-declaration</td>
<td>::= &lt;procedure&gt; &lt;procedure-declaration&gt;</td>
</tr>
<tr>
<td>procedure</td>
<td>::= procedure &lt;identifier&gt; &lt;parameter-list&gt; &lt;variable-declaration&gt; begin  &lt;statement-sequence&gt; end;</td>
</tr>
<tr>
<td>parameter-list</td>
<td>::= parameter &lt;parameter&gt; &lt;parameter-list&gt;</td>
</tr>
<tr>
<td>parameter</td>
<td>::= &lt;identifier&gt; : &lt;parameter-mode&gt;</td>
</tr>
<tr>
<td>parameter-mode</td>
<td>::= value result value-result location name</td>
</tr>
<tr>
<td>statement-sequence</td>
<td>::= &lt;statement&gt; &lt;statement&gt; &lt;statement-sequence&gt;</td>
</tr>
<tr>
<td>declaration</td>
<td>::= declare &lt;identifier-list&gt;; declare &lt;identifier-list&gt; : array;</td>
</tr>
<tr>
<td>identifier-list</td>
<td>::= &lt;identifier&gt; &lt;identifier&gt;, &lt;identifier-list&gt;</td>
</tr>
<tr>
<td>statement</td>
<td>::= &lt;assignment-statement&gt; &lt;input-statement&gt; &lt;output-statement&gt; &lt;call-statement&gt;</td>
</tr>
<tr>
<td>assignment-statement</td>
<td>::= &lt;identifier&gt; := &lt;expression&gt;</td>
</tr>
<tr>
<td>call-statement</td>
<td>::= identifier &lt;expression-sequence&gt;</td>
</tr>
</tbody>
</table>
\<input-statement\> ::= input \<identifier-list\> ;
\<output-statement\> ::= output \<identifier-list\> ;
\<expression-sequence\> ::= \<expression\> | \<expression\> \<expression-sequence\>
\<expression\> ::= \<operand\> | \<expression\> \<operand\>
\<operand\> ::= \<integer\> | \<variable\> | ( \<expression\>
\<variable\> ::= \<identifier\> | \<identifier\> [ \<expression\>
\<identifier\> ::= \<letter\> | \<identifier\> \<letter\> | \<identifier\> ^ \<letter\>
\<integer\> ::= \<digit\> | \<integer\> \<digit\>
\<letter\> ::= A | E | C | D | F | F | G | H | I | J | K | I | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
\<digit\> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
This appendix gives some examples of each implemented Mini-language. A result of each program after being compiled and simulated will be given next to a source code program. Reasonable error messages will be printed out next to a source program if the source program is not correct. The result leading with "Rsl" followed by each output variable and value of that variable will be printed out if the source program is completely correct.
Example C.1 for A Mini-language 'Core'

line:
1 program
2   declare CCOUNT, LIMIT;
3   declare IAST_TERM, THIS_TERM, NEXT_TERM;
4   begin
5     CCOUNT := 0;
6     IAST_TERM := 1;
7     THIS_TERM := 1;
8     input LIMIT;
9
10   while (CCOUNT < LIMIT) loop
11     NEXT_TERM := IAST_TERM + THIS_TERM;
12     IAST_TERM := THIS_TERM;
13     THIS_TERM := NEXT_TERM;
14     CCOUNT := CCOUNT + 1;
15   end loop;
16   output CCOUNT, NEXT_TERM, IAST_TERM, THIS_TERM;
17   end;
18
Start xeq

Rsl 1  COUNT  =  2  
Rsl 1  NEXT_TERM  =  3  
Rsl 1  IAST_TERM  =  2  
Rsl 1  THIS_TERM  =  3  
Finish xeq

Data: 2
Example C.2 for a Mini-language 'Core'

line:
1  declare COUNT **syntax error**, LIMIT;
  **syntax error: line 1: Missing program key word**
2    declare IAST_TERM, THIS_TERM, NEXT_TERM;
3    begin
4      COUNT @ **syntax error**;
5      IAST_TERM := 1;
6      THIS_TERM := 1;
7      input LIMIT;
8
9      ( **syntax error** COUNT < LIMIT)
10     output IASTTERM;
11     NEXT_TERM := IAST_TERM + THIS_TERM;
12     IAST_TERM := THIS_TERM;
13     THIS_TERM := NEXT_TERM;
14     COUNT := COUNT + 1;
15     end loop;
16     end
17
number of errors 3
Example C.3 for a Mini-language 'Core':

```
line:
1 program
2    declare E,T,TT;
3 begin
4    input P,T,TT;
5    output P,T,TT;
6    if (P < 2) then
7        while (T > 2) loop
8            T := T + 1000001;
9            TT := TT + 1;
10       end loop;
11    else
12        E := 4;
13    end if;
14    if (F != 5) then
15        E := 4;
16    endif;
17    output P,T,TT;
18 end;
19
Start xeq
20 Rs1 P = 2
21 Rs1 T = 3
22 Rs1 TT = 0
23 *** Variable out of range ( ; 0 or -2) ***
24
Data: 2 3
```
ExampleC.4 for Mini-language 'Core'

line :
  1 program
  3   begin
  4     while (T ! F) loop
  5       TT := TT - 1;
  6     end loop;
  7     if (T < T) then
  8       E := T;
  9     end if;
 10    while (T!=G) loop
 11      if (I=J) then
 12        if (Y=F) then
 13           while (G=K) loop
 14             A := F;
 15             F := T;
 16           end loop;
 17        else
 18           if (P=U) then
 19             H := T;
 20           end if;
 21       end if;
 22    end if;
 23    end loop;
 24    if (H=F) then
 25       E := U;
 26    end if;
 27  end;
 28
number of errors 1
Example D.1 for a Mini-language 'D'

1 program
-- This program reads in an integer value representing the time
-- on a 24-hour clock and prints out the corresponding 12-hour
clock time. If the input value does not represent a correct
time, the input value is printed.
7 declare TIME, ECURS_AND_MINUTES, HOURS, MINUTES, AM_CF_FM;
8 begin
9 input TIME;
10 HOURS_AND_MINUTES := TIME;
11 HCURS := 0;
12 while (ECURS_AND_MINUTES > 100) loop
13 HOURS AND MINUTES := HOURS_AND_MINUTES - 100;
14 HOURS := HOURS + 1;
15 end loop;
16 MINUTES := HOURS_AND_MINUTES;
17 if (HOURS = 23) then
18 if (HOURS = 24) and (MINUTES = 0) then
19 AM_CF_FM := 0;
20 HCURS := 12;
21 output HOURS, MINUTES, AM_CF_FM;
22 else
23 output TIME;
24 end if;
25 else
26 if (MINUTES > 59) then
27 output TIME;
28 else
29 AM_CF_FM := 0;
30 if (HOURS = 0) then
31 HOURS := 12;
32 else
33 if (HOURS > 11) then
34 AM_CF_FM := 1;
35 if (HOURS > 12) then
36 HOURS := HOURS - 12;
37 end if;
38 end if;
39 end if;
40 output HOURS, MINUTES, AM_CF_FM;
41 end if;
42 end if;
43 end if;
44 end;
Start xeq

Rsl HOURS = 2
Rsl MINUTES = 32
Rsl AM_CR_PM = 1

Data: 1432
Example P.1 for Mini-language Procedure

line:
1 -- Call ty value
2
3 program
4     declare I;
5     declare A : array;
6   procedure SWAP_PY_ICCATION(X: value, Y: value):
7     declare TEMP;
8     begin
9       TEMP := X;
10      X := Y;
11      Y := TEMP;
12  end;
13 begin
14     I := 2;
15     A[I] := 6;
16     output I, A[I];
17     SWAP_PY_ICCATION(I, A[I]);
18     output I, A[I];
19   end;
20
21 Start xeq

Rsl  I = 3
Rsl  I = 3
Finish xeq
Example P.2 for Mini-language 'Procedure'

line:
1   -- Call by result
2
3   program
4       declare I;
5       declare A : array;
6   procedure SWAP_BY_ICCATION(X: result. Y: result):
7       declare TEMP;
8       begin
9           TEMP := Y;
10          X   := Y;
11          Y   := TEMP;
12       end;
13       begin
14          I  := 3;
15          A[I] := epsilon;
16       output I,A[3];
17       SWAP_BY_ICCATION(I,A[I]);
18       output I,A[3];
19       end;
20
21   Start xeq

Rsl  I   = 3
Rsl  I   =
** Error: Attempt to evaluate an undefined variable
Rsl  A[3]    =
** Error: Attempt to evaluate an undefined variable
Finish xeq
Example P.3 for a Mini-language 'Procedure'

line:
  1 -- Call by reference
  2
  3  program
  4    declare I;
  5    declare A: array;
  6  procedure SWAP_FP_ICCATION(X:locati0n, Y: location):
  7    declare TEMP;
  8    begin
  9      TEMP := X;
 10      X := Y;
 11      Y := TEMP;
 12    end;
 13    begin
 14      I := 3;
 16      output I,A[3];
 18      SWAP_FP_ICCATION(I,A[3]);
 19      output I,A[3];
 20    end;
 21
 22  Start xeq

Rsl  I = 3
Rsl  I = 6
Finish xeq
Example P. 4 for a Mini-language "Procedure"

line:
1 -- Passing parameters of different types
2
3 program
4    declare A, S, P;
5    procedure P(C: value, V: result, W: value_result):
6        declare CC;
7        begin
8            CC := e;
9            V := CC + C;
10        end;
11    procedure PP(G: value):
12        begin
13            II := 2 + G;
14        output II;
15        end;
16        begin
17            input A, S;
18            A := A + S;
19        output A, S;
20        B(1, A, S);
21        output A, S;
22        IF(A);
23        end;
24 Start xeq

R sl A = 4
R sl S = 3
R sl A = 6
R sl S = 3
R sl II = 8
Finish xeq
Example T.1 for a Mini-language 'Type

line:
  1 program
  2 declare A, AA: array [3..4] of
  3       array [2..3] of
  4       array [4..6] of
  5       string;
  6 begin
  7       input A[3][3][5];
10      output A[3][2][5];
11      output A[3][2][5];
12      end;
13
start xeq
Rs1 A[3][2][5] 'E+2rat'
Rs1 A[3][3][5] 'CC'
Finish xeq

Data: 'CC'
Example T.2 for a Mini-language 'Type'

line:
1 program
2 declare A, AA: array [3..4] of
3 array [3..3] of
4 array [4..6] of
5 integer;
6 begin
7 input A[3][2][4];
8 input A[3][3][5];
10 output AA[3][2][6];
11 output A[3][3][5];
12 end;
13
start xeq
Rsl AA[3][2][6] 7
Rsl A[3][3][5] 4

finish xeq

Data: 3
4
Example T.3 for a Mini-language 'Type'

line:
  1 program
  2   declare A, AA: array [3..7] of
  3       array [4..7] of
  4       array [2..5] of
  5          boolean;
  6   begin
  7       A[3][5][4] := true and false;
  8       output A[3][5][4];
  9       output AA[3][5][4];
 11      AA[3][6][4] := false;
 12      output A[3][6][4];
 13      output AA[3][6][4];
 14   end;
 15
start xeq
Rsl A[3][5][4] false
Rsl AA[3][5][4] true
Rsl A[3][6][4] true
Rsl AA[3][6][4] false
Finish xeq
**Example 1.4 for a Mini-language 'Type'**

```plaintext
line:
1  program
2    declare A,T: record Y: integer;
3       Z: record
4      F: integer;
5      U: record
6          G: boolean;
7          GG: string;
8                 end record;
9      RR: record
10     TY: integer;
11     YT: boolean;
12     end record;
13    YY: string;
14     end record;
15  end record;
16  begin
17    input A.Z.U.G;
18    input A.Y. A.Z.F;
19    output A.Y;
20    A.Y := 4;
21    A.Y := A.Z.E A.Y;
23  end;
```

```
start xeq
Rsl A.Y 6
Rsl A.Y 12
Psl A.Z.E 8
Rsl A.Z.U.G false
Finish xeq
```

**Data:** false

6

δ
Example 1.5 for a mini-language 'type'

line:
1    program
2      declare A,X,Y :
3          record
4            E : integer;
5            E : string;
6          C : record
7              CC : record
8                DDD : integer;
9                DDE : string;
10              DDF : boolean;
11              end record;
12              RC : boolean;
13              end record;
14      end record;
15      begin
16          A.C.CC.DDF := true and true;
17          if(A.F < A.C.CC.DDD) then
18              output A.C.CC.DDF;
19          end if;
20      A.F := 4;
21      input A.E;
22      A.E := A.F cat A.F;
23      A.E := 'yes' cat 'nop';
24      A.C.CC.DDD := A.E * 12;
25      output A.F, A.C.CC.DDD, A.E;
26      A.C.CC.DDE := A.C.CC.DDE cat A.C.CC.DDE;
27      output A.C.CC.DDE;
28      end;
29      start xeq
30      Rsl A.F  4
31      Rsl A.C.CC.DDD  48
32      Rsl A.E 'yesnop'
33      Rsl A.C.CC.DDF
34      Finish xeq
This appendix shows the example of Mini-language Type. A record type is chosen since it is the most complex one. All functions used to generate the compiler and the interpreter are given. The symbol tables is also printed out to be considered. Before a list of all the files, the structure of each implemented Mini-language for this thesis, shown ir fig C.1, should be considered.
Fig. C.1 Structure used in each implemented Mini-language (in this thesis).
Example Mini-language Type (record type)

File name: tpa6

program
declare A,X,Y :
  record
    F : integer;
    I : string;
    C : record
      CC : record
        DLL : integer;
        DDD : string;
        DDF : boolean;
        end record;
      RC : boolean;
      end record;
  end record;
begin
  A.C.CC.DDF := true and true;
  if(A.F < A.C.CC.DDD) then
    output A.C.CC.DDF;
  end if;
  A.F := 4;
  A.F := 'yes' cat 'nop';
  A.C.CC.DDD := A.F * 12;
  output A.F, A.C.CC.DDD, A.F;
  A.C.CC.DDE := A.C.CC.DDF cat A.C.CC.DDE;
  output A.C.CC.DDE;
end;
Output listing after the above example was executed.

Command:  a.out < traef

line:
1 program
2  declare A,X,Y :
3    record
4      B : integer;
5      E : string;
6      C : record
7        CC : record
8          DDD : integer;
9          DDE : boolean;
10       end record;
11  RC : boolean;
12  end record;
13 begin
14    A.C.C.C.DDF := true and true;
15    if(A.B < A.C.C.C.DDD) then
16      output A.C.C.C.DDF;
17    end if;
18    A.E := 4;
19    A.E := 'yes' cat 'nop';
20    A.C.C.C.DDD := A.E * 12;
21    output A.F, A.C.C.C.DDD, A.E;
22    A.C.C.C.DDF := A.C.C.C.DDE cat A.C.C.C.DDF;
23    output A.C.C.C.DDE;
24 end;

25 start xeq
26  Rsl A.B = 4
27  Rsl A.C.C.C.DDF = 48
28  Rsl A.E = 'yesnop'
29  Rsl A.C.C.C.DDE =
30 Finish xeq
Symbol tables
-------------

File name: idtab.code
-------------

1.) id table (idtab)

<table>
<thead>
<tr>
<th>name</th>
<th>varaddr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>2</td>
<td>Y</td>
</tr>
<tr>
<td>3</td>
<td>T1</td>
</tr>
<tr>
<td>4</td>
<td>T2</td>
</tr>
<tr>
<td>5</td>
<td>T3</td>
</tr>
<tr>
<td>6</td>
<td>T4</td>
</tr>
<tr>
<td>7</td>
<td>T5</td>
</tr>
<tr>
<td>8</td>
<td>T6</td>
</tr>
<tr>
<td>9</td>
<td>T7</td>
</tr>
<tr>
<td>10</td>
<td>T10</td>
</tr>
<tr>
<td>11</td>
<td>T11</td>
</tr>
</tbody>
</table>

File name: var.code
-------------

2.) variable table (vartab)

<table>
<thead>
<tr>
<th>typeaddr</th>
<th>storeaddr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>1</td>
<td>17</td>
</tr>
<tr>
<td>2</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>19</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>21</td>
</tr>
<tr>
<td>7</td>
<td>22</td>
</tr>
<tr>
<td>8</td>
<td>23</td>
</tr>
<tr>
<td>9</td>
<td>24</td>
</tr>
<tr>
<td>10</td>
<td>25</td>
</tr>
<tr>
<td>11</td>
<td>26</td>
</tr>
<tr>
<td>12</td>
<td>27</td>
</tr>
<tr>
<td>13</td>
<td>28</td>
</tr>
</tbody>
</table>
File name: type.code

2. Type Table (typtat)

<table>
<thead>
<tr>
<th>sort</th>
<th>field1</th>
<th>field2</th>
<th>field3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>RECFLD</td>
<td>66</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>INTSUB</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>RECFLD</td>
<td>69</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>STRSUB</td>
<td>773</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>RECFLD</td>
<td>623</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>INTSUB</td>
<td>624</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>RECFLD</td>
<td>622</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>INTSUB</td>
<td>622</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>RECFLD</td>
<td>624</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>STRSUB</td>
<td>77</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>RECFLD</td>
<td>332</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>EOUSUB</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>13</td>
<td>RECFLD</td>
<td>69</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>EOUSUB</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>15</td>
<td>RECFLD</td>
<td>332</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>EOUSUB</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>17</td>
<td>RECFLD</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>18</td>
<td>FOCLEAN</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>FOCLEAN</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>FOCLEAN</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>21</td>
<td>FOCLEAN</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>22</td>
<td>INTEGER</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>23</td>
<td>STRING</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>STRING</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td>STRING</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>INTEGER</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>27</td>
<td>INTEGER</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>28</td>
<td>STRING</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
File name:  . symtab . code

4. symbol table (symtab)

0  4
1 'yesnop'
2  48
3
4  1
5  1
6  0
7
8  0
9
10  1
11  1
12  0
13
14  0
15
16  1
17  1
18  1
19  1
20  1
21  0
22  4
23 'yes'
24 'nop'
25 'yesnop'
26  12
27  48
28
**Quadruple code**

----------

**File name:**  
gq.code

--------

T1 = 1  
T2 = 1  
T3  and  T1  T2  
A.C.CC.DDF = T3  
T4  < A.E  A.C.CC.DDD  
& if  T4  6  
  write  A.C.CC.DDF  
&  goto  
T5 = 4  
A.E = T5  
T6 = `yes`  
T7 = `nop`  
T6  cat  T6  T7  
A.F = T6  
T9 = 12  
T10 * A.E  T6  
A.C.CC.DDE = T10  
  write  A.P  
  write  A.C.CC.DDD  
  write  A.E  
T11  cat  A.C.CC.DDF  A.C.CC.DDE  
A.C.CC.DDF = T11  
  write  A.C.CC.DDE  
stop
/* This program is the main program of a compiler */
/* for Mini-language Type. It does */
/* - read in data file */
/* - call yyparse() for lexical analysis */
/* and syntax analysis */
/* - call xeq to interpret quadruples */
/* - dump out symbol tables for debugging */
/* (using all functions begin with dump) */
/* - report errors if any */
/* (using function "usage") */
/* All variables using ir in this language has a range */
/* of 0 to 99999999. */

#include <stdio.h>
#include "tmain.h"
#include "y.tab.h"
#include "tdec.h"
#include "tquard.h"
#include "tsym.h"
#include "dftype.h"

*/import variables and structures */
extern addr;
extern struct quad *quardsave[MAXQUARD];
extern struct symstore *storetab[PASHSIZE];
extern stack[STACKSIZE], maxstk;
extern nmerrs;
extern scount;
extern ck;
extern struct nlist **keep[PASHSIZE];

main()
{
FILE *obj, *fopen();
int c, i, k, ain;

/* beginning program */

obj = fopen("data", 'r'); /* read in data from data file */
for(k=0; k<10 & &
    fscanf(obj, "%d", &ain) != EOF; k++)
in[k].uval.ival = ain; /* save data in an array */
fclose(obj);
printf(" line : 1 "); /* print line number for input program */

/* get the parser to work */
yyparse();

/* dump cut informatic, if want to */
dumpgen();
dumpsym();
dumpidtable();
dumptype();
dumpvar();

/* Print the errors, if any; continue the interpreter, if not */
if (nmerrs > 0) printf(" number of errors %d0, nmerrs");
else
{
    printf(" start xeq");
dumpgen(); /* write the generated quardruples in a file */
    /* also write symbol table in a file call 'sym.code' */
dumpsym();
    /* execute the input program; if no syntax errors */
    xeq();
dumpsym();
    printf( ");
}

/* print cut my own error message */

,************************chlussung sl,
usage (msg)
char *msg;
{
    extern int nmerrs;
    extern int errormsg;
    if (errormsg == false) {
        printf("**syntax error: line %A: %s 0,lnerr,msg);;
        errormsg = true;
    }
    *msg = NULL;
    return;
}

,***************************
/* yacc error handling */
yyerror (s)
char *s;
{
    extern int en, numline, nmerrs;
    lnrerr = numline + 1;
    en = numline + 1;
    ++nmerrs;
    printf(" **syntax error**");
    return;
}

/* This function makes name for each token or each */
/* assigned constant. Mostly, used in "dump" functions */
/* So it is easier to rebug the programs. */
makerme(name)
int name;
{
    char *str;
    str = (char*) malloc(20);
    switch(name) {
    case INTEGER:  return (strcpy(str,"INTEGER"));
    case TMP: return (strcpy(str,"temp"));
    case IDENTIFIER: return (strcpy(str,"id"));
    case ARRAY:  return (strcpy(str,"ARRAY"));
    case RECORD: return (strcpy(str,"RECORD"));
    case FCTID: return (strcpy(str,"FCTID"));
    case RCFSUP: return (strcpy(str,"RCFSUP"));
    case STRING: return (strcpy(str,"STRING"));
    case FCCLEAN: return (strcpy(str,"FCCLEAN"));
    case INTSUP: return (strcpy(str,"INTSUP"));
    case STFSUP: return (strcpy(str,"STFSUP"));
    case FCSUE: return (strcpy(str,"FCSUE"));
    case SSRANGE: return (strcpy(str,"SSRANGE"));
    case GT: return (strcpy(str,">"));
    case EC: return (strcpy(str,"= ");
    case NFC: return (strcpy(str,"! =");
    case IF: return (strcpy(str,"if ");
    case ITEM: return (strcpy(str,"ITEM"));
    default: return (strcpy(str,"bad"));
}

/* dump 'type' table */
dumptype()
{
    FILE *obj, *fopen();
    struct symtype *prt;
int i;
extern numtype;
obj = fopen("type.code", "w");
fprintf(obj, " type table (typetab) -- ------- 0\n");
fprintf(obj, " sort field1 field2 field3 0\n");
for(i=1; i<numtype; i++)
{
    fprintf(obj, "%d %s ,i, (makerme(typetab[i]->sort));
    fprintf(obj, "%d ,typetab[i]->field1\n");
    fprintf(obj, "%d ,typetab[i]->field2\n");
    fprintf(obj, "%d 0, typetab[i] > field3\n");
}
fclose(obj);

/* dump "var" table */
dumpvar()
{
    FILE *obj,*fcjen();
    struct symvar *prt;
    int i;
    extern numvar;
    obj = fopen("var.code", "w");
    fprintf(obj, " variable table (vartab) --------- \n");
    fprintf(ctj, "variable storeaddr\n");
    for(i=0; i<numvar; i++)
    {
        fprintf(obj, "%ld %d ,i, vartab[i]->typeaddr\n");
        fprintf(obj, "%d ,vartab[i]->storeaddr\n");
    }
    fclose(ctj, "\n");
fclose(obj);
}

/* dump "id" table */
dumpidtab()
{
    FILE *obj,*fcjen();
    struct nlist *ptr;
    int i;
    ctj = fopen("idtab.code", "w");
    fprintf(obj, " id table (idtab) --- --- --- \n");
    fprintf(ctj, " name varaddr\n");
    for(i=0; i<ck; i++)
    {
        fprintf(obj, "%ld %s ,i, keep[i]->name\n");
        fprintf(ctj, "%d ,keep[i]->varaddr\n");
    }
}
/* print symbol table in a file named 'sym.code' */
dumpsym()
{
  FILE *obj, *fopen();
  struct symstore *prt;
  int i;
  obj = fopen("sym.code","w");
  for(i=0;i<count;i++)
  {
    fprintf(obj,"%d ",i);
    prt = storetat[i];
    if (prt->storetype == NULL) fprintf(obj," ");
    else {
      if (prt->storetype == Int
        fprintf(obj,"%d ",prt->ustore.storenum);
      else {
        if (prt->storetype == Str
          fprintf(obj,"%s ",prt->ustore.storeid);
        else fprintf("bad code in dumpsym i %d",i);
      }
    }
  }
  fprintf(obj,"0");
}
fclose(obj);
}
%{
#include "y.tab.h"
#include "th.h"
#include "tdec.h"
YYSTYPE yylval;
}%

Letter [A-Z]+
Digit [0-9]+,
Blank [ ]+

extern list, dec;
int flag = 's';
extern index;
extern numline;
extern firstarray. array, arraysize;

program
{
    ECHC;
    list = norm;
    return PROGRAM;
}

begin
{
    ECHC;
    return START;
}

declare
{
    ECHC;
    dec = true;
    list = iddec;
    return DECLARE;
}

if
{
    ECHC;
    switch (flag)
    {
        case 's':
            return IF;
        case 'e':
            flag = 's';
            return EADIF;
    }
    ECHC;
}

then
{
    ECHC;
    return THEN;
}
else
{
  ECHC;
  return FLSE;
}
record
{
  ECHC;
  switch (flag)
  {
    case 's':
      firstarray = true;
      return RECORD;
    case 'e':
      flag = 's';
      return ENTIRE;
  }
  ECHC;
end
{
  flag = 'e';
  FCHC;
}
input
{
  ECHO;
  list = readin;
  return INPUT;
}
output
{
  ECHO;
  list = writeout;
  return OUTPUT;
}
array
{
  ECHC; array true;
  return ARRAY;
}
cf
{
  FCHC;
  return CF;
}
true
{
  ECHO;
  return TR;
}
false
{
  ECHC; return FS;
}
integer
{
  ECHC;
  return INTEGER;
}
string
{
  ECHO; return STRING;
}
toclean
"\n"
{
  ECHC; return COIGN;
}
"\n"
{
  ECHC; index = true; return CPK;
}
"."
{
  ECHO; index = false; return CPK;
}
"."
{
  ECHC; return DOT;
}
```
"." {
  ECHC;
  return MORE;
}

.";
{
  ECHC;
  switch (flag)
  {
    case 's':
      return FS;
    case 'e':
      flag = 's';
      return ENDPROG;
  }
}

":="
{
  ECHC;
  list = asgn;
  return ASSIGN;
}

"(" {
  ECHC;
  return OPEN;
}

")" {
  ECHO;
  return CIOSE;

"<"
{
  ECHC;
  return IT;
}

">"
{
  ECHO;
  return GT;
}

"="
{
  ECHC;
  return EQ;
}

"!=
{
  ECHC;
  return NEQ;
}

"+"
{
  ECHC;
  return ADD;
}

"-"
{
  ECHC;
  return MINUS;
}
```
"*" { 
  ECHO;
  return MUL;
}

="/" {
  ECHO; return DIV; }
  ECHO; return CAT; }
  ECHO; return AND; }
  ECHO; return CR; }
  strcpy(yylval.id,yytext);
  return STROP; }

{DIGIT}
  ECHO;
  sscanf(yytext, "%d",&yylval.num);
  return INTCP;

{NI} ;

--.*{NI} 
  ECHO;
  numline--; 
  printf(" %d ,numline 1);

??
/*
#include "tquard.h"
#include "thh.h"
#include "th.h"
#include "tdec.h"
#include "tsym.h"
#include <stdio.h>
#include dfitype.h"
#include "type.h"
*/

%start program

%token PROGRAM START ENDFRGM
DECLARE IF THEN
INPUT OUTPUT FINAL EISE
MORE ASSIGN NI PIANK
%token INTEGER IDENTIFIER
%token FS ADD MINUS MUI
EC NEQ IT GT OPEN CICSE
%token INTCP STCF STRING FOOFAN
DIV CAT AND OR COLCA DCT APRAY
CPK CFK CF RECCRT INREC TR FS
INIFX

%right ASSIGN
%left EC NEC
%left IT GT
%left ADD MINUS
%left MUI DIV

% /^ beginning of the program */

program : prog therestofprog
          | gen(STOP=NULL,NULL,NULL);
          |
prog : PROGRAM
         {
          for (i=0; i<MAXID; i++)
          {
            wcsve[i] = C;
          }
E[i] = 1;
}

; error
{ usage('Missing program key word');
}

; therestofprog : declaration

; declaration :
dec idenlist COINC
{}
type 
FS
{
DC = 1;
if($5.num == ARRAY;
 $$ .tp = puttype(ARRAY,(arraysize=poparrsize:)),
 poparrfst()),typearray,;
for(i=0;i:idcount;i++)
for(i=0;i:idcount;i++)
switch ($5.num) {
  case INTEGER: $$ .tp = puttype(INTEGER,where--;MININT,MAXINT
  storetab[scount] = intstore(0);
  $$ .vp[i] = putvar($$ .tp,scount++) ;
  install(idsave[i],$$ .vp[i]);
  break;
  case STRING: $$ .tp = puttype(STRING,where+ ,1,1 );
  $$ .vp[i] = putvar($$ .tp,scount+ );
  install(idsave[i],$$ .vp[i]);
  break;
  case BOCLEAN: $$ .tp = puttype(BOCLEAN,where++,1,1 );
  storetab[scount] = intstore(true);
  $$ .vp[i] = putvar($$ .tp,scount- );
  install(idsave[i],$$ .vp[i]);
  break;
  case RECORD: $$ .vp[i] = putvar($$ .tp,firststore[i]);
  install(idsave[i],$$ .vp[i]);
  break;
  case ARRAY:
  arrstore();
  $$ .vp[i] = putvar($$ .tp,arrfst);
  install(idsave[i],$$ .vp[i]);
  break;
default: printf("bad type in declaration\n"; break;
    } /* case statement */
    } /* for statement */
    }
}
dec : DECLARE
    error ES
    { usage("Missing declare keyword"); };
idenlist : IDENTIFIER
    {
        $$slen = (char*)malloc(strlen($1.id)+1);
        strcpy($$slen,$1.id);
        idsave[idcount++] = $$slen;
        if(dec == false) { dec = true;
            nmerrs +=;
            printf("%s Multiply declsred",$$slen);
        }
    }
    idenlist MORE IDENTIFIER
    {
        $$slen = (char*)malloc(strlen($3.id)-1);
        strcpy($$slen,$3.id);
        idsave[idcount++] = $$slen;
        if(dec == false) { dec = true;
            nmerrs +=;
            printf("%s Multiply declared",$$slen);
        }
    }
    error ES
    { usage("Not an identifier"); };
type : simpletype
    { $$.num = $1.num;
    }
arraytype
    { $$.num = ARRAY;
        popfix();
        EOUNDS = 1;
    }
recordtype
    { $$.num = FECCRD;
firstarray = true;


case INTEGER: typearray = INTEGER;
    $$\cdot$$num = INTEGER;
    }
    }
    STFPING
    $$\cdot$$num = STRING;
    }
    FCCLEAN
    $$\cdot$$num = FCCLEAN;
    }

arraytype : ARFFAY CEK bounds CEK OF
    }
    pushfix(ARRAY);
    arraycount = ;
    typetmp = puttype(SSFANGE,$$.num.0,0);
    if firstarray) { pusharrf(1stypetmp);
        firstarray = false;
    }
    else { FCFAAR = true;
        typetab[poparr()]\rightarrow field2 = typetmp;
    }
    pusharr(1ypearr);,
    pusharr(1ypearr);,
    typetmp;
    }
    extern arrstk[2], errptr;
    if(notfix) { poparr(); notfix = false; }
    TYPE = $$7.$$num;
    while (morearray) {
        switch(TYPE) {
        case integer: typearray = INTEGER;
            $$\cdot$$t\ = puttype(1ntSUE,INTEGER,\poplb\,),\pcpub\,
typetab[poparr()]\rightarrow field2 $$\cdot$$tp;
            morearray = false; break;
        case STRING: typearray = STRING;
            $$\cdot$$tp = puttype(STRSUFPSTFPING,\poplb\,),\popub\,
typetab[poparr()]\rightarrow field3 = $$\cdot$$t\;
            morearray = false; break;
        case FCCLEAN: typearray = FCCLEAN;
            $$\cdot$$t\ = puttype(POSUF,CCLEAN,\poplb\,),\popub\,;
```c
typedef [tmp=pcparr] field2 = $$morearray = false; break;
case ARRAY: TYPE = typearray;
    break;
case RECCRD: typearray = RECORD;
arraystore = arraystore / EOUNDS;
$$tp = puttype(RECSUE,$7.tp, popl$, popub() type[poparr] ->field2 = $$tp;
morearray = false;
break;
} /* case */
} /* while */
if(changearysize) pusharrsize(arraysize);
changearysize = false;
arraysize - 1;
firstarray = true;
morearray = true;
};

bounds : INTCF DCT DCT INTO:
{
    pushlb($1.num);
pushub($4.num);
$$num = $4.num - $1.num + 1;
EOUNDS = $$num;
arraystore = $$num * arraysize;
arraystore = $$num * arraystore;
lb[DC] = $1.num;
ub[DC] - $4.num;
++DC;
for(i=1;i<DC;i++)
    D[i] = EOUNDS * D[i];
changearysize = true;
}

recordtype : reckey
{
    if(wc > 0 & typeptr > -1) {
        wcsve[wc-1] = wordcount;
        popsave[rc+] = pcptype();
    }
    temp = wc;
    temp = temp + nut;
    wordcount = wcsve[temp];
}
recseq
ENTREC
{```
else firstrec[fr] = numtype-1;
push0type(numtype-1); push0type(numtype-1);

CCION
{}
type ES
{
switch ($5.num) {
  case INTEGER:
    $$tp = puttype(INTSUB,where+,MINIM,MAXIM);
    putstore($5.num);
    break;
  case FCLEAN:
    $$tp = puttype(POSUB,where-.1,1);
    putstore($5.num);
    break;
  case STRING:
    $$tp = puttype(STRSUB,where+.1,1);
    putstore($5.num);
    break;
  case RECCHI: break;
  case ARRAY:
    $$tp = puttype(ARRAY,(POUNDS=poparrsize()),
                 poparrfst(),typearray);
    where = where + POUNDS +1;
    break;
  default: print("bad type in rec0");
            break;
}
if(typeptr > -1 && $5.num != RECORD) {
  tmp = pop0type();
typetab[tmp]->field2 = $$tp;
}
if(changearrsize) pusharrsize(arraysize);
changearrsize = false;
arraysize = 1;
/* for the grammar */

content : econtent
  statseq
  endprog
;
econtent : START
  !
  error ES { usage(\"Missing \"tegr\" key word\") ; }
;
endprog : ENTPRCG
error
{ usage("Missing end");  }

statseq : statseq stat

stat : ifstat

ncrifstat : assignstat

outputstat

assignstat :

variable ASSIGN express ES
{
  $$slen = (char*)malloc(strlen($1.slen) 1);
  strcpy($$slen,$1.sler);
  switch(typtab[$1.tp]->sort) {
  case INTEGER: case INTSUP:
    if(typtab[$2.tp]->sort != INTEGER &&
        typtab[$3.tp]->sort != INTSUP)
    { printf(" Type combination");
        errcnt++;
    } else
        printf(" ASSIGN,strelem($3.slen),NULL,
                strelem($$.slen)");
        break;
  case STRING: case STRSUP:
    if(typtab[$2.tp]->sort != STRING &&
        $2.tp != STRING &&
        $3.tp != STRSUP &&
        typtab[$3.tp]->sort != STFSUP)
    { printf(" Type combination");
        errcnt++;
    } else
        printf(" ASSIGN,strelem($3.slen),
                NULL,strelem($$.slen)");
        break;
  case BCCIFAN: case ECSUP:
    if(typtab[$3.tp]->sort != BCCIFAN &&
        typtab[$3.tp]->sort != ECSUP)
    { printf(" Type combination");
    } errcnt;
  }
else
  \texttt{gen}(\texttt{ASSIGN}, \texttt{strelem($\varphi$.slen)}, \texttt{NULL}).
  \texttt{strelem($\varphi$$.slen)});
  break;
  default: \texttt{printf(' wrong opcode type');}
  \}
  list = \texttt{norm}:
  }

  \texttt{variable error ES}
  \{} \texttt{usage.'Assign operator expected \':='}; \} ;

\texttt{ifstat : firstif therestofif}

\texttt{firstif : IF} \texttt{express THEN}
  \{
  int \textit{i};
  \texttt{elseflag[0]} = \texttt{true};
  \texttt{if} (\texttt{elseflag[\textit{i}]} == \texttt{true}) { \texttt{\textit{i}++};
  \texttt{elseflag[\textit{i}]} = \texttt{false};
  \texttt{endifflag[\textit{i}]} = \texttt{false};
  \}
  \texttt{push(addr)};
  \texttt{gen(If, strelem($\$$.slen), intelem(addr 1), NULL)};
  +\texttt{ifcount[\textit{i}]};
  
\texttt{therestofif : endifpart}

\texttt{! statseq ELSE}
  \{
  int \textit{tmp}p;
  int inttmp;
  \texttt{elseflag[\textit{i}]} = \texttt{true};
  \texttt{if} (\texttt{endifflag[\textit{i}]} == \texttt{true})
  \texttt{tmp p = pop(!)};
  \texttt{fixup, intelem(addr+1), pop()}, 4);
  \texttt{if} (\texttt{endifflag[\textit{i}]} == \texttt{true})
  \texttt{push(tmp p)};
  \texttt{push(addr)};
  \texttt{gen(GOTO, NULL, NULL, strelem(NULL))};
  \}
  \texttt{endifpart}

\texttt{error ES} \{} \texttt{usage('Forgot 'end if;' keyword')} ; \} ;
endifpart : statseq endkey
endkey : FNTIF ES
{
  int in temp, j, i;
  if( --if count[ii] == ∅ ) { /* ∅ to higher level */
    gen(GCTO,NUII,NUII,
       (intelem(inttemp-addr 1)));
    if( else flag[ii] == true { /* fix else */
      else flag[ii] = false;
    if(max stk > 0)
      fixup(intelem(inttemp),pop(),4);
    }
    if(max stk > 0)
      /* fix if */
    fixup(intelem(inttemp),pop(),4);
    ii--;
    if(ii == ∅ & max stk > ∅)
      /* clear if stack at end */
    while(max stk > ∅)
      fixup(intelem(inttemp),pop(),4);
    else {
      if( else flag[ii] == true ) {
        else flag[ii] = false;
        if(max stk > 0)
          fixup(intelem(addr),pop(),4);
      }
      else fixup(intelem(addr),pop(),4);
      if(end if flag[ii] == true ) {
        end if flag[ii] = false;
        if(max stk > 0)
          fixup(intelem(addr),pop(),4);
      }
      push(addr);
      error(GCTO,NUII,NUII,strelem(NUII);
    }
    end if flag[ii] = true;
  }
  ! error ES
  { usage("Forgot 'end if;' keyword");
  }

; input stat : INPUT varlist ES
  { list = norm; }
INPUT varlist error ES

    usage("Missing ';'");

; outputstat : OUTPUT
    varlist ES
    { list = norm; }
;
    CUTFPUT varlist error ES
    { usage("Missing ';'"); }
;
varlist : varlist MORE variable
    {
        $$slen = (char*)malloc(strlen($$slen),1);
        strcpy($$slen,$$slen);
        switch(listsave)
            { }
    }
variable
    {
        $$slen = (char*)malloc(strlen($$1.slen,1));
        strcpy($$slen,$$1.slen);
        switch(listsave) };
express : opseq operand
    {
        list = norm;
        strtemp = newtemp ;
gen($1.num, strelem($1.slen), strelem($2.slen),
    strelem(strtemp));
$$ .slen = strtemp;
if($1.num == GT || $1.num == AND || $1.num == CR || $1.num == EQ || $1.num == NEQ || $1.num == LT)
{
    $$ .tp == puttype(ECCIFAN, 1. MININT, MAXINT);
    storetab[scount+1] = intstore(true);
    $$ .place = scount - 1;
    install($$.slen, putvar($$.tp, $$ .place));
}
if($1.num == CAT)
{
    $$ .tp = puttype(STRING, 1.0, 0);
    storetab[scount+1] = strstore(NULL);
    $$ .place = scount - 1;
    install($$.slen, putvar($$.tp, $$ .place));
}
if($1.num == ADD || $1.num == MUL || $1.num == DIV
    $1.num == MINUS
{
    $$ .tp = puttype(INTEGER, 1. MININT, MAXINT);
    storetab[scount+1] = intstore(0);
    $$ .place = scount - 1;
    install($$.slen, putvar($$.tp, $$ .place));
}
"

cperand
    {$$ = $1; };

opseq
    : operand operator
    {
    $$ .slen - (char*)malloc(strlen($1.slen)+1);
    strcpy($$.slen, $1.slen);
    $$ .num = $2.num;
    }

opseq operand operator
    {
    list = norm;
    strtemp = newtemp();
    gen($1.num, strelem($1.slen), strelem($2.slen),
        strelem(strtemp));
    $$ .slen = strtemp;
    if($1.num == MUL || $1.num == DIV || $1.num == ADD || $1.num == MINUS
    {
        $$ .tp = puttype(INTEGER, 1. MININT, MAXINT);
        storetab[scount+1] = intstore(0);
        $$ .place = scount - 1;
gen(NU1, strelem(strtemp), intelem(T[DC]),
    strelem(strtempM));
DC++;
if(notfirst) {
    strtemp = newtemp();
    $$\cdot tp = puttype(INTEGER,1,MININT,MAXINT);
    storetab[scount++] = intstore(\);$$\cdot place = scount -1;
    install(strtemp, putvar($$. tp,$$. place));
    gen('ADD, strelem(strtmpsv), strelem(strtemp));
    strtmpsv = strtemp;
}
else strtmpsv = strtempM;
notfirst = true;
$$\cdot slen = strtmpsv;
} /* PK */
else {
    list = norm;
    strtemp = newtemp();
    gen(ASSIGN, intelem($1.num), NUL1, strelem(strtemp));$$\cdot slen = strtemp;
    $$\cdot tp = puttype(INTEGER,1,MININT,MAXINT);
    storetab[scount] = intstore($1.num);
    $$\cdot place = scount -1;
    install($$. slen, putvar($$. tp,$$. place));
}
fig = $1.num;

; STFCP
{
    list = norm;
    strtemp = newtemp();
    gen(ASSIGN, strelem($1.id), NUL1, strelem(strtemp));$$\cdot slen = strtemp;
    $$\cdot tp = puttype(STRING,1.1.1);
    storetab[scount++] = strstore(NULL);
    $$\cdot place = scount +1;
    install($$. slen, putvar($$. tp,$$. place));
}

; CPFN express CIOUSF
{
    $$ = $2;
}

operator :
IT { $$\cdot num = IT; }
EC { $$\cdot num = EC; }
variable : variablertmp
{
  idfirst = true;
  rumplace = 0;
  switch($$.num) {
    case RECORD:
      $$ .place = get();
      if(!notfound) {
        printf(" Undefined variable ");
        nmerrs++; 
        return(NULL);
      }
      $$ .tp = savetype;
      $$ .slen = (char*)malloc(strlen($$.slen)+1);
      strcpy($$.slen,$$.slen);
      isave = $$ .slen;
      break;
    case ARRAY:
      strtmpsv = strtemp;
      strtemp = newtemp();
      $$ .tp = puttype(INTEGER,1,MININT,MAXINT);
      strlen$=scount++ = irtstore(555555);
      $$ .place = scount -1;
      install(strtemp,putvar($$.tp,$$.place));
      getflag$ = true;
      tmpt = get();
      getflag = false;
      error(ASSIGN.intelem(tmp),strelm(NULL),
            strelm(strtemp));
      error(ARRAY,strelm(strtemp),strelm(strtmpsv),
            strelm(strtemp)=newtemp());
      break;
    default:
      break;
  }
}
{  
  icsave = (char*)malloc,strlen($1.slen) +1);  
  strcpy(icsave,$1.slen);  
}  
$$.$slen = strtemp$;  
$$.$t$= puttype(ITEM, 1, MININT, MAXINT);  
stcretat[$\text{scoun}$++$] = intstore(2);  
$$.$place = $\text{scoun}$ -1;  
install($$$.$slen.putver($$$.$tp,$$$.$place));  
$$.$tp=typetab[typetab[$$.$tp]-fie$Z$]-fie$Z$  
break;  
default:  
$$.$slen = (char*)malloc(strlen($1.slen)+1);  
strcpy($$$.$slen,$1.slen);  
iosave = $$.$slen;  
break;  
} /* case */  
testfirst = true;  
cin = 0;  
DC = 1;  
item$\text{cp}$count=0;  
nctfirst = false;  
  

\textbf{variabletmp} : \textbf{variabletmp DOT IDENTIFIER}  
{  
  if(testfirst) {  
    vars$\text{cp}$[cin] = (char*)malloc(strlen($1.slen)+1);  
    strcpy(vars$\text{cp}$[cin+$]$,1.slen);  
    testfirst = false;  
  }  
  vars$\text{cp}$[cin] = (char*)malloc(strlen($2.$id)+1);  
  strcpy(vars$\text{cp}$[cin-$1$,2.$id$);  
  $$.$slen = (char*)malloc(strlen($1.slen)$  
  strlen($2.$id$)+2);  
  sprintf($$$.$slen, "2s[2d] ",$1.slen,$2.$id$;  
  $$.$\text{rum} = \text{RFCCRD};  
}  

\textbf{variabletmp CPK}  
{  
  \textbf{PK} = true;  
}  

\textbf{express CPK}  
{  
  \textbf{PK} = false;  
  if(list$\text{cp}$save==writ$\text{cp}$out :  
    list$\text{cp}$save=read$\text{cp}$r  
      {  
      $$.$slen = (char*)malloc(strlen($1.slen)  
      sprintf($$$.$slen, "2s[2d] ",$1.slen,$2.$id$+1);  
    }  
  }  
}
else $$\$.slen = $4$.slen;
idtemp = newtemp();
$$\$.tp = puttype(INTEGR,1,MININT,MAXINT);
storetab[scount+ ] = intstore($5$5$5$5$5$);
$$\$.place = scount -1;
$$\$.num = ARRAY;
if(fig < lb[DC-2] || fig > ub[DC-2])
{
    printf(" ** Index out of bounds");
    nerrs+ ;
}$$\$.tp = savetype;
}

IDENTIFYFF
{
    listsave = list;
    if(testfirst) {
        if('install($1.id,1)=NULI)
        return('NULI);
    }
    $$\$.slen = (char*)malloc(strlen($1.id)+1);
    strcpy($$.slen,$1.id);
    vartmp = idtab[hash($1.id)]->varaddr;
    $$\$.place = vartab[vartmp]->storable;
    $$\$.tp = vartab[vartmp]->typeaddr;
    savetype = $$\$.tp;
    $$\$.num = rorm;
    varsave[0] = (char*)malloc(strlen($1.id)+1);
    strcpy(varsave[0],$1.id);
    }

toclean : TR
    { $$\$.num = true; }
| FS
    { $$\$.num = false; }
}
# define PROGRAM 267
# define START 269
# define ENIPRCG 269
# define DECLAR 260
# define IF 261
# define THEN 267
# define INPUT 263
# define OUTPUT 264
# define EALIF 265
# define ELSE 266
# define MORE 267
# define ASSIGN 268
# define NI 269
# define BLANK 270
# define INTEGER 271
# define IDENTIFIER 272
# define ES 273
# define ADD 274
# define MINUS 275
# define MUI 276
# define EQ 277
# define NEQ 278
# define LT 279
# define GT 280
# define CFEN 281
# define CICSI 282
# define INTCF 283
# define STCF 284
# define STRING 285
# define EOOFIEN 286
# define LIV 287
# define CAT 288
# define AND 289
# define CR 290
# define COICN 291
# define LTC 292
# define ARRAY 293
# define CPK 294
# define CPF 295
# define CF 296
# define RECORD 297
# define ENTREC 298
# define TR 299
# define ES 300
# define INDEK 301
**This file contains all the two-stacks functions used in *, */
** in Mini-Language Type compiler (mostly, in type.y). */

```c
#define MAXTYPE 30

int typesstk[MAXTYPE], typeptr=0, maxtype=0;
int aristk[MAXTYPE], arrptr=0, maxarr=0;
int ubstk[MAXTYPE], ubptr=0, maxub=0;
int lstk[MAXTYPE], lbptr=0, maxlb=0;
int afstk[MAXTYPE], afptr=0, maxaf=0;
int aszstk[MAXTYPE], aszptr=0, maxasz=0;
int fixstk[MAXTYPE], fixptr=0, maxfix=0;

pushfix(val)
int val;
{
    int i;
    extern idcount;
    if(maxfix+1 > MAXTYPE) printf(" over flow fix stack");
    fixstk[fixptr++] = val;
    return;
}

int popfix()
{
    int i, inttemp;
    if(fixptr < -1) printf(" underflow fix stack");
    inttemp = --fixptr;
    maxfix--;
    return(fixstk[inttemp]);
}

pushtype(val)
int val;
{
    int i;
    if(maxtype+1 > MAXTYPE) printf(" over flow type stack");
    typesstk[typeptr++] = val;
```
```c
return;

int pop_type()
{
    int i, int temp;
    if (typeptr < -1) printf(" underflow type stack");
    int temp = --typeptr;
    maxtype--;  
    return (typestk[inttemp]);
}

/**
 * pusharr(val)
 *  
 * int val;
 *  
 *     if (maxarr++ > MAXTYPE) printf(" over flow array stack");
 *     arrstk[arrptr++] = val;
 *     return;
 */

int poparr()
{
    int int temp;
    int temp = --arrptr;
    maxarr--;  
    return (arrstk[inttemp]);
}

/**
 * pushlb(val)
 *  
 * int val;
 *  
 *     if (maxlb++ > MAXTYPE) printf(" over flow lb stack");
 *     lbstk[ltptr++] = val;
 *     return;
 */

int poplb()
{
    int int temp;  int temp = --lbptr;  maxlb--;  
    return (lbstk[inttemp]);
}

/**
 * pushub(val)
 *  
 * int val;
 *  
 *     if (maxub++ > MAXTYPE) printf(" over flow ub stack");
 */
```
popub()
{
    int inttemp; inttemp = --ubptr;  
    return(ubstk[inttemp]);  
}

pusharrf
/
***************  
pusharrf
/
***************  
}

pusharrsize(val)
int val;
{
    if(maxasz > MAXTYPE)
        printf("overflow arrsize stack");
    aszstk[aszptr++] = val; return;
}

poparrsize()
{
    int inttemp; inttemp = --aszptr;  
    return(aszstk[inttemp]);  
}

/
***************  
pusharrsize(val)
int val;
{
    if(maxasz > MAXTYPE)
        printf("overflow arrsize stack");
    aszstk[aszptr++] = val; return;
}

poparrsize()
{
    int inttemp; inttemp = --aszptr;  
    return(aszstk[inttemp]);  
}

/
***************  
pusharrsize(val)
int val;
{
    if(maxasz > MAXTYPE)
        printf("overflow arrsize stack");
    aszstk[aszptr++] = val; return;
}

poparrsize()
{
    int inttemp; inttemp = --aszptr;  
    return(aszstk[inttemp]);  
}

/
***************  
pusharrsize(val)
int val;
{
    if(maxasz > MAXTYPE)
        printf("overflow arrsize stack");
    aszstk[aszptr++] = val; return;
}

poparrsize()
{
    int inttemp; inttemp = --aszptr;  
    return(aszstk[inttemp]);  
}

#include "tsym.h"
#include "tquerd.h"
#include "thh.h"
#include "tdec.h"
#include "y.tab.h"
#include "th.h"
#include "stdio.h"

#define RECEIE 62
#define true 1
#define false 0
#define NUII 0
#define MAXVAI 100

/* import variables */
extern nmerrs, list;
extern dec;
extern dumptype();

int ck = 0;

/**********************
/* "hash" generates hash value */

hash(s)
char *s;
{
    register int hashval;
    for(hashval = 0; *s ; s++)
    {
        hashval <<= 7;
        hashval += (unsigned) *s;
    }
    return hashval % HASHSIZE ;
}

/**********************/
struct nlist *lockup(s);    /* lock for s in symbol table */
char *s;
{
    struct nlist *rp;
    for (np = idtab[hash s]); np != NUII; np = np->next)  
        if (strcmp(s,np->name)==0)
            return(np);    /* found it */
    return (NUII);            /* not found */
/***************/
int trace = 0;
struct nlist *install(name, usage) /* put in hash table */
char *name;
int usage;
{
    struct nlist *p, *lookup();
    char *strsave(), *malloc();
    int *intsave();
    int hashval;

    if ((np = lookup(name)) == NUII) { /* not found */
        if (list == asgn || list == readin || list == writeout, {
            nmerrs;
            printf(" Undefined identifier: %s 0, name);
            return (NUII);
        }
        np = (struct nlist *)malloc(sizeof(*np));
        if (np == NUII)
            { printf(" np");
            return (NUII);
        }
        if ((np->name - strsave(name)) == NUII)
            { printf(" strsave ");
            return (NUII);
            }
        np->varaddr = usage;
        hashval = hash(np->name); /* produces starting index in */
        np->next = idtab[hashval]; /* array hashtable */
        idtab[hashval] = np;
        keep[c++] = np;
    }
    else
    {
        switch (list) {
            case iddec:
                dec = false;
                break;
            case asgn:
                break;
        }
    }
    return (np);
}

/*********************/
strcmp (s, t) /* return < 0 if s < t, 0 if s=t, > 0 if s > t */
char *s, *t;
{
    int i;
    i = 0;
    while (s[i] == t[i])
        if (s[i++] == '\0')
            return (s[i]-t[i]);
    return (s[i]-t[i]);
}

,********************
char *strsave(s)
char *s;
{
    char *p;
    if ((p = nalloc(strlen(s)+1)) != NULL)
        symcpy(p, s);
    return (p);
}

,********************
symcpy(s, t) /*copy t to s*/
char *s, *t;
{
    while (*s++ = *t++)
        ;
    return;
}

,********************
#define AIICCSIZE 1000
static char allocbuf[AIICCSIZE];
static char *allocp = allocbuf;

char *nalloc(n) /* for 'strsave' function */
int n;
{
    if (allocp + n == allocbuf + AIICCSIZE) {
        allocp -= n;
        return (allocp - n);
    } else
    {
        return(NULL);
    }
}
int puttype(sort, field1, field2, field3)
int sort, field1;
int field2;
int field3;
{
    extern numtype;
    struct symtype *rew;
    new = (struct symtype*)malloc(sizeof(struct symtype));
    new->sort = sort;
    new->field1 = field1;
    new->field2 = field2;
    new->field3 = field3;
    typetab[numtype++] = new;
    dupmtype();
    if(numtype > MAXTYPE)
        printf("Type space is exhausted. Expand MAXTYPE in tsym.h");
    return (numtype-1);
}

int putvar(typeaddr, store)
int typeaddr;
int store;
{
    struct symvar *new;
    extern numvar;
    new = (struct symvar*)malloc(sizeof(struct symvar));
    new->typeaddr = typeaddr;
    new->storeaddr = store;
    vartab[numvar++] = new;
    return(numvar-1);"
/* This file contains functions that will put in a */
/* value in symbol tables; vartab, storetab, typetab. */

#include "tstore.h"
#include "y.tab.h"
#include <stdio.h>

#define true 1
#define false 0

extern sccunt; extern firststore[MAXID];
extern isave; extern iisave; extern j,i;

int i, ij;

struct symstore *irststore(par)
int par;
{
    struct symstore *temp;
    temp = (struct symstore*)malloc(sizeof(struct symstore));
    temp->storetype = Int;
    temp->ustore.storenum = par;
    return(temp);
}

struct symstore *strstore(par)
char *par;
{
    struct symstore *temp;
    temp = (struct symstore*)malloc(sizeof(struct symstore));
    temp->storetype = Str;
    temp->ustore.storeid = (char*)malloc(strlen(par)+1);
    strcpy(temp->ustore.storeid, par);
    return(temp);
}

putstore(type)
int type;
{
    int ii, k;
    extern int idcount;
    extern first: extern typearray;
    extern arraystore; extern array;
    int PCUNDS;
    extern struct symstore *irtstoreO;
    extern struct symstore *strstore(v;
    POUNDS = arraystore;
    if(first == false)
        for(k=0;k<idcount;k++) {
            switch (type) {
                case INTEGER:
                    for(ii=0;ii<POUNDS;ii++)
                        storetab[scount++] = intstore(0);
                    break;
                case STRING:
                    for(ii=0;ii<POUNDS;ii++)
                        storetab[scount++] = strstore(NUII);
                    break;
                case ECCLEAN:
                    for(ii=0;ii<PCUNDS;ii++)
                        storetab[ii+i] = intstore(true);
                    break;
                default: break;
            } /* case */
            firststore[i] = i;
            first = true;
        }
    else {
        jj = 0;
        isave = (scount-1) / idcount;
        iisave = iisave + PCUNDS;
        while(jj < idcount-1, {
            for(i=scount-1;i>iisave;i++)
                storetab[i+POUNDS] = storetab[i];
            switch (type) {
                case INTEGER:
                    for(ii=0;ii<PCUNDS;ii++)
                        storetab[ii+i+1] = intstore(0);
                    break;
                case STRING:
                    for(ii=0;ii<BCUNDS;ii++)
                        storetab[ii+i] = strstore(NUII);
                    break;
                case ECCLEAN:
                    for(ii=0;ii<POUNDS;ii++)
                        storetab[ii+i] = intstore(true);
                    break;
                default: break;
            }
        })
    }/* else */
} /* case */
```c
} /* case */
firststore[jj] = i+2;
iseave = i+1 liseave;
scount = scount + BLOCKS;
} /* while jj*/

/* fix the least one */
switch (type) {
  case INTEGER:
    for(ij=E;ij<FOUNDS;ij++)
      storetab[scount-1] = intstore(0);  break;
  case STRING:
    for(ij=8;ij<FOUNDS;ij++)
      storetab[scount+1] = strstore(NULL);
    break;
  case BOOLEAN:
    for(ij=5;ij<FOUNDS;ij++)
      storetab[scount++] = intstore(true);
    break;
  default: printf("bad putstore\n";  break;
} /* case2 */
return;
  */

/*******/
/* This function deals with array. */
/*******/

arrstore()
{
  int i;
  extern scount, arrfst, arraysize, TYPE;
  for(i=0;i<arraysize;i++)
  { switch(TYPE) {
    case INTEGER:  storetab[scount++] = intstore(0);  break;
    case STRING:  storetab[scount++] = strstore(NULL);  break;
    case BOOLEAN:  storetab[scount-1] = intstore(true);  break;
    default:  printf("wrong arrstore\n");
  }
  if(i == 0) arrfst = scount-1;
  }
}


```c
#include "tquard.h"
#include "thh.h"
#include "tdec.h"
#include "tsym.h"
#include "y.tab.h"
#include th.h"
#include 'dftype.h'
#include <stdio.h>

extern savetype;
extern *varsave[MAXID];
extern cin;
int notfound;
int vc;
int keepplace;
extern numplace;
extern getflag;

/* This output of this function gives the address */
/* (index to storetab array) variable. */

int get()
{
    struct symtype *nnp;
    struct symvar *mp;
    int strp, i;

    mp = varat[tabl[hash(varsave[0])]]->varaddr);
    keepplace = mp->storeaddr;
    if(getflag)
    {
        getflag = false;
        return(keepplace);
    }

    nnp = typetab[np->typeaddr];
    if(nnp->sort == INTEGER || nnp->sort == STRING ||
       nnp->sort == PCLEAN || nnp->sort == ITEM ||
       nnp->sort == INTSUP || nnp->sort == STRSUP) {
        savetype = nnp->typeaddr;
        numplace - 1;
        return (keepplace);
    }

    else {
        notcurd = true;
        vc = 0;
        keepplace = getplace(nnp,mp);
        if(vc != cin-1) notcurd = true;
    }
}
```
struct symvar *mmp;

/* "getplace" is called recursively until a simple type */
/* is found and then return the index of storetab to */
/* get. */

int getplace(rnp, mp)
    /* return pointer to the variable */
struct symvar *mp;
struct symtype *nr;
{
    extern notfound;
    int n, i;
    char *s;

    switch (nnp->scrt) {
    case INTEGER: case STRING: case BOOLEAN:
    case ITEM: /* array case */
        return (keepplace);
    case INTSUP: case STSUE: case FOSUF:
        sub = (nnp->field1);
        keepplace = (mp->storeaddr).sub;
        return (keepplace);
    case RECORD:
        nnp= nnp->field1;
        mmp = mp;
        if((nnp->field1 = hash(varsav[e[++vc]])) {
            notfound = false;
            if(typeat[nnp->field3]->sort == INTSUP ||
                typeat[nnp->field2]->sort == STSUE ||
                typeat[nnp->field2]->sort == FOSUF
                saveType = nnp->field3;
                if((keepplace=getplace(typeat[nnp->field3], mp), = NULL)
                    return (NULL);
            else
                return (keepplace);
        }
        break;
    case ARRAY:
    }
savetypetypeat[nrp->field2]->field3;
breaek;
case RECORD:
mnp = mp;
notfound = true;
for(i=ci;i<nnp->field1 & & notfound;i++)
    keepplace = getplace(typeat[nnp->field1],m1);
    if(keepplace == NULL) return (NULL);
    else
        return(keepplace);
breaek;
default:
    printf(" wrong sort ");
    printf(" nrp->sort %d",nrp->sort);
breaek;
}
#include "y.tat.h"
#include "thh.h"
#include "tquad.t"
#include "tsym.h"
#include "tdec.h"
#include <stdio.h>

/* import variables */
extern struct symquad *symint();
extern struct nlist install();
extern itempcount;
extern list;

/* variables used in this file */
int maxstk = 0;
int addr = 0;
int tempcount = 0;
int stack[STACKSIZE], stackptr = 0;
int whilestack[WSIZE], whileptr = 0;
struct quardruple *quardsave[MAXQUARD];

/************************/
/* Function generating the quardruples for the input program. */

gen(op, arg1, arg2, rsl)
int op;
struct quardruple *arg1, *arg2, *rsl;
{
    struct quardruple *new; /* create temporary area */
    new = (struct quardruple*) malloc(sizeof(struct quardruple));
    new->Op = op;
    new->Arg1 = arg1;
    new->Arg2 = arg2;
    new->Res = rsl;
    quardsave[addr++] = new; /* save each quardruple */
    if (addr > MAXQUARD) /* make sure won't be any bus error */
        printf("Not enough space, expand MAXQUARD in 'quad.h' 0");
    return;
}

/****************************/
/* in case any argument in quardruple is an integer */

struct quardruple *intelem(arg)
int arg;
{
    struct quard *temp;
    temp = (struct quard*)malloc(sizeof(struct quard));
    temp->type = Int;
    temp->term.numval = arg;
    return(temp);
}

/**************************/
/* in case the argument is a string */
struct quard *strelem(arg)
char *arg;
{
    struct quard *temp;
    temp = (struct quard*)malloc(sizeof(struct quard));
    temp->type = Str;
    temp->term.idval = (char*)malloc(strlen(arg)+1);
    strcpy(temp->term.idval, arg);
    return(temp);
}

/**************************/
/* function to create a temporary variable */
char *newtemp()
{
    char *str;  /*create temporary name */
    tempcount++;
    list = norm;
    str = (char*)malloc(IDMAX 1);  /* allocate space to it */
    sprintf(str, Tp%d " ,tempcount);
    return(str);
}

/**************************/
/* function for generating a temporary name "T1...Tn" */
char *newindtmp()
{
    char *str;
    tempcount++;
    str = (char*)malloc(IDMAX 1);
    sprintf(str, "T%d " ,tempcount);
    return(str);
}
/***************/
/* finishinf the backpatching by fixing up those saved addresses */

fixup(value,qudstack,pst)
struct quard *value;
int qudstack, pst;
{
    switch(pst)
    {
        case 2 : quardsave[qudstack]-Arg1 - value; break;
        case 3 : quardsave[qudstack]-Arg2 = value; break;
        case 4 : quardsave[qudstack]-Rsl = value; break;
    }
}

,***************
/* function to push an address of any quardruple, which has to be fixed */
/* later, on a stack called 'stack'.

push(val)
int val;
{
    maxstk++;
    stack[stackptr++] = val;
    return;
}

,***************
/* function to pop each quardruple cut off stack */

int pop()
{
    int inttempi;
    if (stackptr = -1) printf('stack IF error');
    inttemp = --stackptr;
    maxstk--;
    return(stack[inttemp]);
}

/***************
/* the same idea as function 'push', but this is for loop statement */

pushwh(val)
int val;
{
    whilestack[whileptr-] = val;
}
/***************/
/* another pop function for loop statement */

int popwh()
{
    if (whileptr == 0) printf("stack WHILE error");
    return(whilestack[--whileptr]);
}

/***************/
/* In case of any curiosity, the file called gq.code can be locked to */
/* fix any problems in generating quadrant files. */
dumpgen()
{
    FILE *obj, *fopen();
    int i = 0;
    struct quad *prt;
    char *a = "", *otj;
    obj = fopen("gq.code", "w");
    while (i < addr)
    {
        prt = quardsave[i] - Rs1;
        if (prt == NULL) fprintf(obj, "%s", a);
        if (prt->type == Int) fprintf(obj, "%d", prt->term.numval);
        if (prt->type == Str) fprintf(obj, "%s", prt->term.idval);
        fprintf(obj, "%s", (makestr(quardsave[i]->Op)));
        prt = quardsave[i]->Arg1;
        if (prt == NULL) fprintf(obj, "%s", a);
        if (prt->type == Int) fprintf(obj, "%d", prt->term.numval);
        if (prt->type == Str) fprintf(obj, "%s", prt->term.idval);
        prt = quardsave[i]->Arg2;
        if (prt == NULL) fprintf(obj, "%s", a);
        if (prt->type == Int) fprintf(obj, "%d", prt->term.numval);
        if (prt->type == Str) fprintf(obj, "%s", prt->term.idval);
        fprintf(obj, "%s", a);
        if (prt == NULL) fprintf(obj, "%s", a);
        if (prt->type == Int) fprintf(obj, "%d", prt->term.numval);
        if (prt->type == Str) fprintf(obj, "%s", prt->term.idval);
        fclose(obj);
    }
}

/*********************/
/* This function generate names for 'dump' functions so */
/* it is easier to understand. */

makestr(name)
int name;
{
    char *str;
}
str = (char*) malloc(20);
switch(name) {
    case ASSIGN: return (strcpy(str,"=")) ;
    case GOTO: return (strcpy(str,"goto"));
    case MINUS: return (strcpy(str,"-"));
    case ADD: return (strcpy(str,"+
    case MUL: return (strcpy(str,"*"));
    case LT: return (strcpy(str,"<"));
    case GT: return (strcpy(str,">
    case EQ: return (strcpy(str,"=
    case NEQ: return (strcpy(str,"!=
    case CAT: return (strcpy(str,"cat");
    case AND: return (strcpy(str,"and
    case OR: return (strcpy(str,"or")
    case IF: return (strcpy(str,"if")
    case IN: return (strcpy(str,"read");
    case CUT: return (strcpy(str,"write");
    case STCP: return (strcpy(str,"stop");
    case ARRAY: return (strcpy(str,"[]");
    default: return (strcpy(str,"bad");
# include <stdio.h>
#include "tsym.h"
#include "tdec.h"
#include "y.tab.h"
#include "thb.h"
#include "dftype.h"
#include "tquad.h"
#include tsim.h
#define MAXID 20
#define PASHSIZE 787

/* Import variables and structures */
extern struct symquard *symint;
extern struct nlist *symtat[PASHSIZE];
extern struct nlist *install();
extern struct symquard *value;
extern addr;
extern struct quad *quardsave[MAXQUARD];
extern int in[10];
extern numvar;
char *varsave[MAXID];
char *ITEMA, *ITEME;

struct symstore *AP;
int cin, A, F, errors=0, iq=0;
int item = false;
int arg=place, arg2place, rslplace;
int rs1type, arg1type, arg2type;

/* startirg execution */
xeq()
{
    FILE *obj, *fopen();

    /* Temporary variables used here */
    int typesave[MAXTYPE], tt. count=0, k=0, j. temp;
    /****

    /* Used in GRCUP */
    extern get();
    extern getvar();
    extern savetype;
    extern struct symtype *tyetat[MAXTYPE];
    extern getflag;
    /****

    getflag = false;

T = "true";   F = "false";

/* start from the first quadruple */

while(iq<addr) {
    temp = quardsave[iq]- Cp;
    switch (temp) {
        case ASSIGN: GROUP3;
            temp = typetab[rslttype]->sort;
            switch(typetab[rslttype]->sort) {
                case STRING: GROUP3;
                    if(*ARG1ID == "'")
                        /* case 'string' */
                        strsim(rslplace,ARG1ID);
                    else {
                        GROUP1;
                        strsim(rslplace,STARG1ID);
                    }
                    break;
                case INTEGER: case INTSUP: case FOCIEAN: case POSUF:
                    if(ARG1MP==Int) intsim(rslplace,ARG1NM);
                    else {
                        GROUP1;
                        intsim(rslplace,STARG1NM);
                    }
                    break;
                case ITEM:
                    GROUP1;
                    if('SCRT1='ITEM)
                        { /* SCRT1 ITEM */
                            GROUP3;
                            if(ITEMETP= Str)
                                strsim(STSHOTM,ITEM1ID);
                            else
                                intsim(STSHOTM,ITEM1INM);
                        }
                    else
                        { /* SCRT1 ITEM */
                            if(SOFT1==STRING ;  'SCRT1=STPSUP)
                                strsim(STSINM,STARG1ID);
                            else
                                intsim(STSINM,STARG1NM);
                        }
                    break;
            default: printf("bad type in ASSIGN");
                    }
                    item = false;
                    break;
        case ARRAY:
            item = true;
            math();
            STFSITP = Int;
STPSINM = A + B;

break;

case ADD:
    math();
    STRSITP = Int;
    STPSINM = A + B;
    break;

case MINUS:
    math();
    STRSITP = Int;
    STPSINM = A - B;
    break;

case MUL:
    math();
    STRSITP = Int;
    STPSINM = A * B;
    break;

case DIV:
    math();
    STRSITP = Int;
    STPSINM = A / B;
    break;

case CAT:
    GFCUP1;  GRCUP2;  GPCUP3;
    if (SCRT1 == STRING || SCRT1 == STRSUP | SCRT1 == ITEM) &
    (SCRT2 == STPING | SCRT2 == STRSUP | SCPT2 == ITEM))
    {
        if (SCRT1 == ITEM) ITEMA = ITEM1ID;
        else ITEMA = STAEG1ID;
        if (SCRT2 == ITEM) ITEMA = ITEM2ID;
        else ITEMA = STARG2ID;

        /* test if both are null string */

        if (*ITEMA ! = '' & *ITEME ! = '')
        else {
            string = (char*)malloc(strlen(ITEMA)
                         + strlen(ITEME) + 1);
            STRSITP = string;
            sprintf(string, "%.ITEMA.ITEME;
                                mystr copy(STRSITP,string);
            STRSITP = Str;
        }
    }
    else usage("variable type string required");
    break;

case LE:  GRCUP1;  GRCUP2;  GPCUP3;
    GTOPCCDE();
    if (A < P) intsim(rslplace,true);
    else intsim(rslplace,false);
    break;

case GE:  GRCUP1;  GRCUP2;  GROUP3;
    GTOPCCDE();
    if (A > P) intsim(rslplace,true);
    else intsim(rslplace,false);
    break;

case FC:  GRCUP1;  GRCUP2;  GROUP3;
    GTICFCDE();
    if (A == P) intsim(rslplace,true);
    else intsim(rslplace,false);
    break;

case NE:  GRCUP1;  GRCUP2;  GROUP3;
    GTICFCDE();
    if (A != P) intsim(rslplace,true);
    else intsim(rslplace,false);
    break;
case CR: GFCUP1; GFCUP2; GROUP3;
GETCPOCTE();
  if(A == false && 1 == false) intsim(rslplace,false);
else intsim(rslplace,true);  break;
case AND: GFCUP1; GROUP2; GFCUP3;
GETCPOCTE();
  if(A == true && P == true) intsim(rslplace,true);
else intsim(rslplace,false);  break;
case IF:
  GROUP1;
  if(STARG1NM == true) iq = quardsave[iq]->Ar;2->term.numval;
  if(STARG1NM == false) iq = quardsave[iq]->Rsl- term.numval-
  break;
case GOTO:
  iq = quardsave[iq]->Rsl-.term.numval-1;
  break;
case IN:
  GFCUP1;
  if(item) {
    P = SCRT1;
    iq = iq -1;
    GRCUF2;
    A = STRSINM;
    iq = iq + 1;
  }
else B typetat[argv1type] sort;
typesave[count++] = F;
aa = (char*)malloc(strlen('20')+1);
obj = fopen("data","r");
for(j=0;j<count-1 && errors -- 0;j++)
  { 
    if(typesave[j]=-INTEGER || typesave[j]=-INTSUE)
      fscanf(obj,"%d",&bb);
    else /* it is a toolean or string type */
      fscanf(obj,"%s",aa);
    }
  if(typesave[count-1]=-INTEGER || typesave[count-1]=-INTSUE
    /* data type is an integer */, 
    if(item)
      fscanf(ctj,"%g",&ITEMRSINM);
    else fscanf(obj,"%d",&STARG1NM);
  else /* data type is a string or boolean */ 
      fscanf(ctj,"%s",ae);
    if(typesave[(count-1)= STRING
        || typesave[count-1]= STESUE)
        /* data type is a string */
      if('test(aa) == true
         if item) {
        ITEMRSIID = (char*)malloc(strlen(aa) 1);
strcpy.ITEMSID, aa;
}
else {
STARG1ID = (char*)malloc(strlen(aa)+1);
strcpy(STARG1ID, aa);
}
else { /* data type is boolean */
if(strcmp(aa, T"=.0")) {
if(item) ITEMRSINM = true;
else STARG1NM = true;
}
else if(strcmp(aa, T"=.2")) {
if(item) ITEMRSINM = false;
else STARG1NM = false;
}
else { errors++; 
printf("*** Illegal data ***");
break;
}
}
fclose(obj);
item = false;
break;
case OUT:
if(errors == 0){
printf( Fsl Qs - ",,quardsave[iq].Arg1-"term.idval);
GROUP1;
if(item) {
F = SCRT1;
iq = iq - 1;
GROUP2;
AA = storeatat[STRSINM];
iq = iq + 1;
}
else {
AA = storeatat[arg1place];
F = typetat[arg1type]->sort;
}
if(F != FCOILEAN | | B == FOPEN)
if(AA-> ustore.storenum == 1) printf("true");
else printf("false");
exeqprint(AA);
item = false;
break;
}

case STOP:
printf( 'Finish xeq F); return;
default : printf("Can not find cyclede");
iq+;
}
}

/*****************************/
/* just to make sure that the collect values are printed out */

xegprint(prt)
struct symstore *ptr;
{
    if(prt-:storetype == Int,
        printf(" %d",prt->ustore.storenum);
    else {
        if(prt-:storetype == Str,
            printf("%s",prt->ustore.storeid);
        else printf(" type %d",prt->storetype);
        if(prt == NULL) return;
    }
}

/*****************************/
/* Just to make sure that I will get only the string, get rid */
/* of '*' in between each string which occurs when operator */
/* 'CAT' is used. */

mystrcpy(s,t)
char *s,*t;
{
    *s = ' ';
    while(*t != ' ')
        { if(*t != ' ')
            *s++ = *t++ ;
            else t++;
        }
    *s++ = ' ' ;
    *s = ' ';
}

/*****************************/
/* Test each input datum when STRING type is required. */
/* Return NULL if it is not a string, 'true' if it is. */

test(s)
char *s;
{
    if(*s++ != ' ') / * check the first character */
        printf( ' *** illigal date ***' ; errors' ;

return (NULI);  
}
while(*s != '\n'); /* scan until the last character */
if (*s-1) != '\n') {
    printf(' *** illegal data ***'); errors++;  
    return (NULI);  
}
return (true);

/******************************/
/* read a variable and put each character group in a 'varsave' array */
getvar(t);
char *t;

extern char *varsave[20];
int i, count=0;
char *s;
cin = 0;
s = (char*)malloc(strlen(t)+1); /* get space for s */
while(*t != '\0') {  
    if (*t != ' ' && *t != '[') { count++;  
* s++ = *t++;  
    } else {  
        t++;  
* s++ = ' ';  
for (i=0; i<count; i++) s--; /* go back to the first character  
varsave[cin] = (char*)malloc(strlen(s)+1);  
strcpy(varsave[cin],s);  
s = (char*)malloc(strlen(t)+1);  
count = 0;  
    }
} /* for the last group */
*s = ' ';  
for (i=0; i<count; i++) s--;  
varsave[cin] = (char*)malloc(strlen(s)-1);  
strcpy(varsave[cin+1],s);  
return;

/******************************/
/* for ADD, MINUS, MUL, DIV */

math()
{
    if(ARG1TF == Str) {
GROUP1;
    if(SCPT1==ITEM)
{
  A = storetat[STARG1NM] -> ustore.storenun;
} else
  A = STARG1NM;
else A = APG1NUM;
if ARG2TP == Str {
  GROUP2;
  if (SCR12 != ITEM) {
    F = storetat[STARG2NM] -> ustore.storenun;
  } else
    F = STARG2NM;
} else
  F = ARG2NUM;
GROUP3;

*************************************************************************
/* This procedure acts as a define statement */

GETOPCODE();
{
  /** check at argument1 **/
  if (SCR11 == ITEM) {
    if (ITEM1TP != Int) { PADMESS; errors++; }
    else A = ITEM1NM;
  } else
    if (STARG1TP != Int) { PADMESS; errors++; }
    else A = STARG1NM;

  /** look at the argument2 **/
  if (SCR12 == ITEM) {
    if (ITEM2TP != Int) { PADMESS; errors++; }
    else F = ITEM2NM;
  } else
    if (STARG2TP != Int) { PADMESS; errors++; }
    else F = STARG2NM;

  int sim(val1, val2)
  int val1;
  int val2;
  { 
    storetat[val1] > storetype = Int;
  }
storetat[val1]->ustore.storenum = val2;

strcmp(val1.val2, int val1, char *val2, 
{
    storetat[val1]-. stcretype = Str;
    storetat[val1].ustore.storeid = (char*)malloc(strlen(val2)+1);
    strcpy(storetat[val1]->ustore.storeid, val2);
}
#define Int 300
#define Str 910
#define MAXID 20
#define HASHSIZE 737

struct symstore {
    int stcretype;
    union {
        int storerum;
        char *storeid;
    } ustore;
};
struct symstore *stcretat[HASHSIZE];
#define Int 600
#define Str 10
#define MAXID 20
#define HASHSIZE 767

struct symstore {
    int storetype;
    union {
        int storenum;
        char *storeid;
    } ustore;
};

struct symstore *storetab[HASHSIZE];
```c
#define STACKSIZE 1024
#define MAXIATA 20
#define false 0
#define true 1
#define IICMx 10

int lnerr = 0;
int nerror = 0;
int errormsg = false;
int en;
int numline = 0;
char *idsave[IDCMX];

struct {
    int datatype;
    union {
        int ival;
        char *strval;
    } uval;
} in[MAXDATA];
#define readin 700
#define writeout 710
#define iddec 720
#define rcmd 730
#define asgn 740
#define putin 750
struct quad{
    int type;
    union {
        int numval;
        char *idval;
    } term;
};

struct aquad{
    int Cp;
    struct quad *Arg1;
    struct quad *Arg2;
    struct quad *Res1;
};

#define MAXQUARD 1000
#define Int 900
#define Str 910
#define LIMAX 20
#define STACKSIZE 1000
#define WSIZE 1000
#define SFACE (printf("", ));
#define Int 00
#define Str 910
#define TMP 20
#define HASHSIZE 767
#define MAXTYPE 1000
#define MAXIL 20

struct nlist {    /* id-table entry */
    char *name;
    int varexpr;
    struct nlist *next;    /* next entry in chain */
};
struct nlist *idtab[HASHSIZE], *keep[HASHSIZE];

struct symvar {
    int typeaddr;
    irt storeaddr;
};
struct symvar *vertab[MAXTYPE];

struct symtype {
    int sort;
    int field1;
    int field2;
    int field3;
};
struct symtype *typetab[MAXTYPE];

struct symstore {
    int storetype;
    union {
        int storenum;
        char *storeid;
    } ustore;
};
struct symstore *storetab[HASHSIZE];
#define SSRANGE 950
#define RECEIT 960
#define INTSUP 990
#define STSUP 995
#define PCSUP 999
#define R1CSUP 999
#define ITEM 955
#define NULI 0
#define MAXINT 999999999
#define MININT 0
#define on 0
#define off 1
typedef struct {
    char *id[MAXID];
    int num;
    char *slen;
    int tp;
    int vj[MAXID];
    int place;
} irs;
#define YYSTYPE irs

tmp
#define true 1
#define false 0

#define AFG1NUM quardsave[iq]- Arg1-.term.numval
#define AFG2NUM quardsave[iq]- Arg2- term.numval
#define AFG1TF quardsave[iq]- Arg1->type
#define AFG2TF quardsave[iq]- Arg2- type
#define AFG1ID quardsave[iq]- Arg1->term.idval
#define AFG2ID quardsave[iq]- Arg2->term.idval
#define FSLID quardsave[iq]- Rsl->term.idval
#define FSLTP quardsave[iq]- Rsl->type
#define STESPIM storetab[rslPLACE]- ustore.storenun
#define STAFG1IM storetab[arg1place]- ustore.storeun
#define STARG2NM storetab[arg2place]- ustore.storeun
#define STESP1D storetab[rslPLACE]- ustore.storeid
#define STARG1ID storetab[arg1place]- ustore.storeid
#define STARG2ID storetab[arg2place]- ustore.storeid
#define STAPG1P storeat[ar1place] storetype
#define STAPG2P storeat[ar2place] storetype
#define SCRT1 typetab[ar1type] sort
#define SCRT2 typetab[ar2type] sort
#define SCRTSI typetab[rs1type] sort
#define GRCUP1 getvar(ARG1IT : ar1place get(); ar1type = savetype;
#define GRCUF2 getvar(ARG2IT : ar2place = get'); ar2type = savetype;
#define GRCUF3 getvar(P3IIIT rslplace = get(); rsltype = savetype;
#define ITEM1ID storetab[STARG1AM] ustore.storeid
#define ITEM2ID storetab[STARG2AM] ustore.storeid
#define ITEM1NM storetab[STARG1AM] ustore.storenum
#define ITEM2NM storetab[STARG2AM] ustore.storenum
#define ITEMRS1NM storetab[STRSINM] ustore.storenum
#define ITEMRS2NM storetab[STRSINM] ustore.storeid
#define ITEMRS1P storetab[STRSINM] storetype
#define ITEM1P storetab[STARG1AM] storetype
#define ITEM2P storetab[STARG2AM] storetype
#define EADMESS printf "Comparison statement, integer variable is needed external not found; /* check variable, get from get.c */
extern int get();
extern struct symstore *intstore();
extern struct symstore *strstore();
extern putvar();
extern puttype();
extern struct nlist *install();
extern struct nlist *idtab[MAXSIZE];
extern struct symtype *typetab[MAXTYPE];
extern struct quard *quardsave[MAXQUAD];
extern maxstk, exter; stackptr; exter addr; extern errors;
extern pcotype(); extern pushtype(); int PCPAPE false;
extern poplt(); extern pushlt(); extern popub();
extern poparr(); extern pusharr(); exter arptr;
extern pusharrfst(); extern pcarrfst();
extern pusharrsize(); extern pcarrsize();
extern typesstk[20]; extern typetkr; extern maxtype;
extern fixstk[20]; exter fixptr; extern maxfix;
extern dumotype();

struct quard *irtelem(), *strelem();

char *varsave[MAXII];
char *strtemp, *rewtemp();
char *rewindtmp();
char *idsave[MAXID], *string[MAXID];
char *strptrs, *strtemp;
char *iosave;
char *idtemp;

int tmp;
int arraystore = 1;
int numplace;
int D[MAXID], DC = 0;
int XX;
int fig;
int itemplace = 0;
int FK;
int rotfix = true;
int getflag = false, notfirst = false;
int listsave = 0;
int rub = 0, temp;
int firsttmp, chainrmp, ad;
int list = norm;      /* check variable in tsym.c */
int first = false;    /* in putstore */
int where = 0;        /* where in storesut */
int savetype;
int dec, i;
int elseiflag[MII], endifflag[MII], ifcount[MAXIF];
int ii;
int idcount = 0;
int ut[MAXID], lt[MAXID], arraysize = 1, wordcount;
int numtype = 1, numvar = 0, sccount = 0;
int firststore[MAXID], save;
int cin = 0, st;
int index = false, ind;
int testfirst = true;
int popsave[MAXID];
int wcsve[MAXID];
int firstvec[MAXID];
int fr = -1, rc = 0, wc = 0;
int isave = 0, isave, j, i = 0;
int morearray = true, firstarray = true, TYPE, typetmp, array;
int FCOUNTS = 1, array = false;
int changearray = false;
int arrayve[20], ac = 0, arraycount = 0;
int idfirst = true;
typedef struct {
    char *id[MAXID];
    int num;
    char *slen;
    int tp;
    int vp[MAXID];
    int place;
    } irs;
#define YYSTYPE irs
int vartmp;
#define true 1
#define false 0

#define ARG1NUM quardsave[iq]->Arg1->term.numval
#define ARG2NUM quardsave[iq]->Arg2->term.numval
#define ARG1TF quardsave[iq]->Arg1->type
#define ARG2TF quardsave[iq]->Arg2->type
#define ARG1ID quardsave[iq]->Arg1->term.idval
#define ARG2ID quardsave[iq]->Arg2->term.idval
#define RSLID quardsave[iq]->Rsl->term.idval
#define FSITF quardsave[iq]->Rsl->type
#define STRSINM storetab[rslplace]->ustore.storenum
#define STARG1NM storetab[ar1place]->ustore.storenum
#define STARG2NM storetab[ar2place]->ustore.storenum
#define STIRSLID storetab[rslplace]->ustore.storeid
#define STARG1ID storetab[ar1place]->ustore.storeid
#define STARG2ID storetab[ar2place]->ustore.storeid
#define STIRSTF storetab[rslplace]->storetype
#define STARG1TF storetab[ar1place]->storetype
#define STARG2TF storetab[ar2place]->storetype
#define SCRT1 typetab[ar1type]->sort
#define SCRT2 typetab[ar2type]->sort
#define SCRTSRL typetab[rsltype]->sort
#define GROUP1 getvar(ARG1ID); ar1place = get(); ar1type = savetype;
#define GPCUP2 getvar(ARG2ID); ar2place = get(); ar2type = savetype;
#define GROUP3 getvar(FSITF); rslplace = get(); rsltype = savetype;
#define ITEM1ID storetab[STARG1NM]->ustore.storeid
#define ITEM2ID storetab[STARG2NM]->ustore.storeid
#define ITEM1NM storetab[STARG1NM]->ustore.storenum
#define ITEM2NM storetab[STARG2NM]->ustore.storenum
#define ITEMRSINM storetab[STRSINM]->ustore.storenum
#define ITEMRSIT storetab[STRSINM]->ustore.storeid
#define ITEMSTF storetab[STRSINM]->storetype
#define ITEM1TF storetab[STARG1NM]->storetype
#define ITEM2TF storetab[STARG2NM]->storetype
#define EATMESS printf(" Compare statement, Integer variable is needed");
extern notfound;    /* check variable, get from 'get.c' */
extern int get();
extern struct symstore *intstore();
extern struct symstore *strstore();
extern putvar();
extern puttype();
extern struct nlist *ininstall();
extern struct nlist *ictab[HASHSIZE];
extern struct symtype *typetab[MAXTYPE];
extern struct aquard *quardsave[MAXQUARD];
extern maxstk, exter; stackptr; extern addr; extern nerrs;
extern pctype(); extern pushtype();
extern poplit(); extern pushlit(); extern pophub(); extern pushub();
extern poparr(); extern pusharr(); extern arrptr;
extern pusharrfst(); extern pcarrfst();
extern pusharrsize(); extern pcarrsize();
extern typestk[20]; extern typeptr; extern maxtype;
extern fixstk[20]; extern filptr; extern maxfix;
extern dumptype();

struct quad *intelem();
struct varstore[MAXID];
char *varsave[MAXID];
char *strtemp, *newtemp();
char *newirdtm();
char *idsave[MAXID], *string[MAXID];
char *strtmpsv, *strtemp;
char *iosave;
char *idtemp;

int tmp;
int arraystore = 1;
int numplace;
int L[MAXID], DC = 0;
int XX;
int fiq;
int itemcount = 0;
int PK;
int nctfix = true;
int getflag = false, notfirst = false;
int listsave = 0;
int rub = 0, temp;
int firstmp, chairmp, ed;
int list = norm;    /* check variable in tsym.c */
int first = false;    /* in putstore */
int where = 0;    /* where in storesub */
int savetype;
int dec, i;
int elseflag[MII], endifflag[MII], ifcount[MAXII];
int ii;
int idcount=0;
int ub[MAXIC], lb[MAXIC], arraysize=1, wordcount;
int numtype = 1, numvar = 0, sccurt = 0;
int firststore[MAXIC], save;
int cin = 0, st;
int index = false, ird;
int testfirst=true;
int po$psave[MAXID];
int wcsve[MAXIC];
int firstrec[MAXIC];
int fr = -1, rc = 0, wc = 0;
int iisave = 0, iseve, j=0;
int morearray=true, firstarray=true, TYPE. typearray, type$mp. arrfst;
int FOUND$=1, array=false;
int changearray=false;
int arrayve[20], ac=0, arraycount=0;
int idfirst=true;
This guide is written for someone that is already familiar with the Unix operating system.

Each compiler was kept in a different directory as follows:

<table>
<thead>
<tr>
<th>Name of Mini-language</th>
<th>Directory name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>Core\</td>
</tr>
<tr>
<td>D</td>
<td>D\</td>
</tr>
<tr>
<td>Type</td>
<td>Type\</td>
</tr>
<tr>
<td>Procedure</td>
<td>proc\</td>
</tr>
</tbody>
</table>

The various directories are accessed by typing

```
cd Directory-name
```

for example

```
cd Core
```

will get you into the 'Core' directory.

The 'a.out' of each directory contains object code which is the compiler for the Mini-language. The command

```
a.out < program1
```

will compile program1 which is a program in the Mini-language of that directory. The result will be given on the
screen. The data for each user program must be given in a file called 'data'. The 'data' file is reserved in every directory to be a data file, since the compiler will read data from a read file only. Each datum in 'data' file may be separated by a 'blank' or a 'newline'.

The Output

What will be shown on the screen after the command

a.out < program1

is entered, is the listing of program1, the error messages, if any, at each point an error occurs and the message

number of error ?

if, for example, there are three errors. If there are not any errors the message shown will be

start xeq

which means the execution starts at the point. The interpreter is called and will interpret each quadruple until the last one ('stop'). The answer given next might be in the form

Rsl  AFO  =  5

This means that the result value of variable 'AFO' is an
integer 5, or

\[ Rsl \quad APC \quad 'true' \]

which means the result value of variable 'APC' is a string 'true', or

\[ Rsl \quad APC \quad = \quad true \]

which means the result value of variable 'APC' is a boolean value of 'true'.

### 7.1 Additional Comment

Each array or stack used to generate each of the compilers were given a certain amount of space (number). When a message such as

'Not enough space, expand MAXCUIAR in 'quard.h'

appears on the screen, it means that the constant number given, 'MAXCUIAR' in this case, is too small or the user program is too big. There are two ways that this can be remedied:

(1) Go to the file 'quard.h' and change the 'MAXCUIAR' value to a bigger one.

(2) Make the user program smaller.
An Implementation of Four of Ledgard's Mini-languages

by

Piyanai Saowarattitada

A thesis, submitted to The Faculty of the School of Computer Science and Technology, in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

June 29, 1983

Approved by:

Thesis Advisor                John L. Ellis                Date

Committee Member                John A. Biles                Date

Committee Member                Chris Comte                Date