1983

An implementation of four of Ledgard's mini-languages

Piyanai Saowarattitada

Rochester Institute of Technology
School of Computer Science and Technology

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

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

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 Piyanai 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.

ABSTRACT

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 Ledgard 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 language 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, D 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, triple, C programming language, regular expression, token, VAX 11/780, UNIX.

TABLE OF CONTENTS

LIST OF FIGURES

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 Ledgard 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 existing

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 D-structures, D for Dijkstra [as in Bruno 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 : LALR(1) 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. Iex 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 Iex and Yacc. The generated Iex 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.

CHAPTER 2

COMPILERS

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 implemertation, 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 INTEJER 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 DO 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.

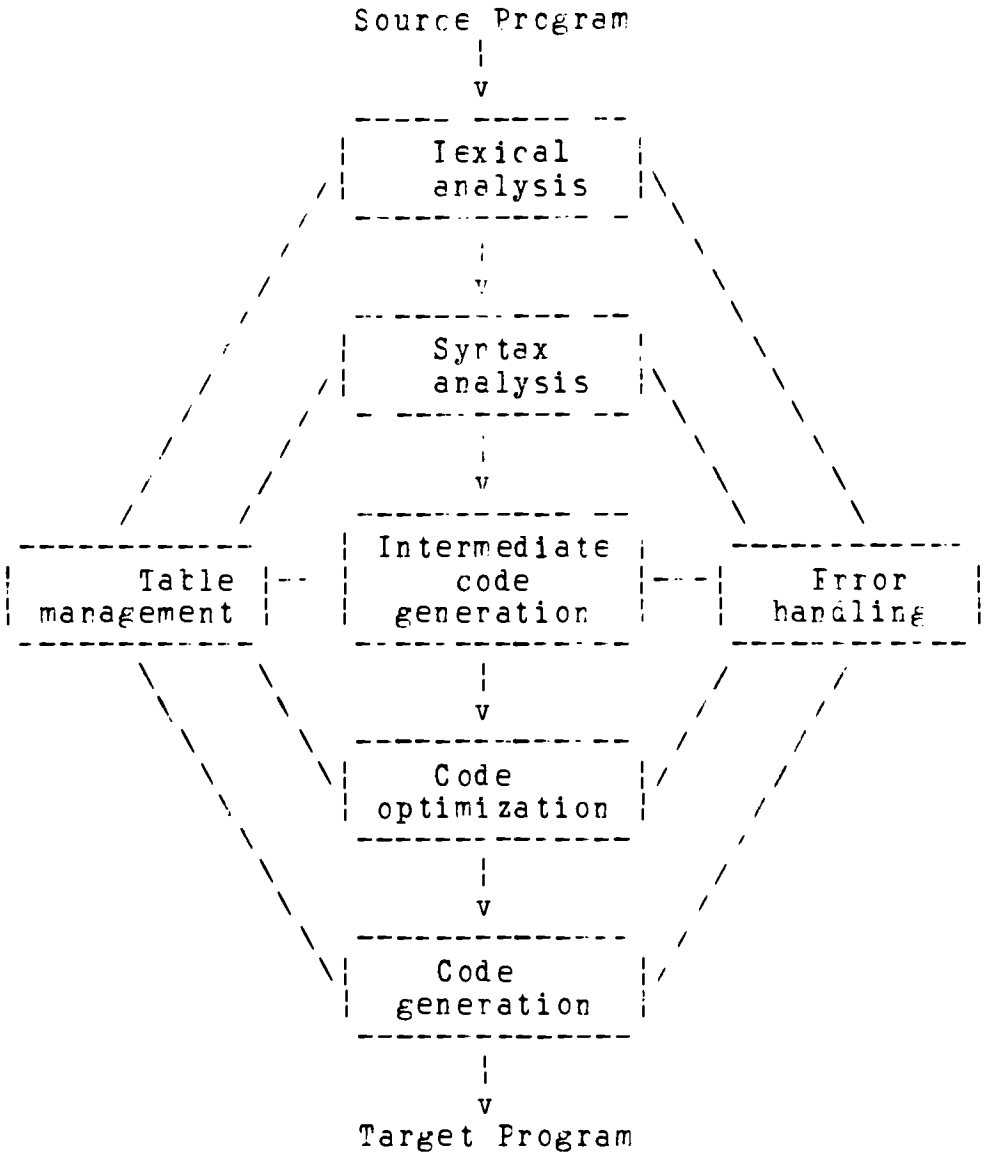


Fig 2.1 Phases of a compiler.

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 LEX - A LEXICAL ANALYZER GENERATOR

Lex is one of the tools used to help in lexical analysis of text. Lex 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, Lex can write code in different host languages. At present, there is only one host languages, 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 Lex.

Lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible

at each input point. If necessary substartial lockahead is performed on the input, tut the input stream will be tacked up to the end of the current partition.

2.1.1 Iex Source

The general format of Iex source is as follows :

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

The absolute minimum Iex program (in theory) is

```
%%
```

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

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

```
a 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. Regular

expressions in Iex use the following operators:

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

for example, the rule

```
program         printf("found keyword program");
```

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

Iex 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.

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

```
program          keyword actions..;
[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 automator which executes an LR 1) 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                    : may be empty.
%%
rules
%%
program                         : 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 Miri-language Core to give a general idea of Yacc:

```
program         :     PROGRAM
                       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         :     DECLARE idenlist ES

                ;

idenlist        :     idenlist IDENTIFIER
                |     IDENTIFIER

                ;

        .... next rules ...
```

Fig 2.2 Example Yacc for Miri 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

```
declare     :     DECLARE idenlist FS
```

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

```
idenlist    :     idenlist IDENTIFIER
            |     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) Based on 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
        PROGRAM  shift 3
           error

        program  goto 1
        prog  goto 2

state 1
        $accept :  program_$end

        $end  accept
           error
```

```
state 2
        program :  prog_$$1 therestofprog
        $$1 : _     (1)

        .  reduce 1

        $$1  goto 5

state 3
        prog :  PROGRAM_     (3)

        .  reduce 3


state 4
        prog .  error_     '4'

        .  reduce 4


state 5
        program :  prog $$1_therestofprog

        error  shift 10
        DECIARE  shift 9
        .  error

        therestofprog  goto 6
        declaration  goto 7
        dec  goto 8

state 6
        program :  prog $$1 therestofprog_     (2

        .  reduce 2


state 7
        therestofprog :  declaration_content

        error  shift 14
        START  shift 13
        .  error

        content  goto 11
        econtent  goto 12

state 8
        declaration :  dec_idenlist COLON $$6 type $$7 ES

        error  shift 17
```

```
            IDENTIFIER  shift 16
            .  error

            idenlist  goto 15

        state 9
            dec :  DECLARE_     (9)

            .  reduce 9


        state 10
            dec :  error_ES

            ES  shift 18
            .  error


        state 11
            therestofprog :  declaration content_     (5)

            .  reduce 5

          ... next state ...
```

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 Iex, 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 Iex with Yacc

Iex is designed to simplify interfacing with Yacc. What Iex generates is a program named "yylex()", the name required by Yacc for its analyzer. Normally, the default main program on the Iex library calls this routine, but if Yacc is loaded, and its main program is used, Yacc will call "yylex()". In this case each Iex 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 Iex 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 UNIX command sequence can just be :

```
yacc core.y
lex core.l
cc y.tab.c -ll
```

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 gererated by a 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 notation, syntax trees, quadruples and triples. Fig 2.4 compares each type by showing how the expression:

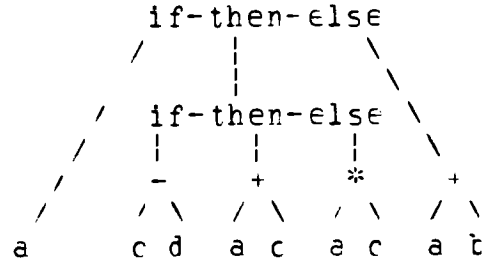
if a then if c=d then a+c else a*c else a+b

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, Op, ARG1, ARG2, and RSI (result). The Op 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 Rsl 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.

TYPE	FORM	NOTE
Postfix Form	acd-ac+ac*?at+?	Using ? as a ternary postfix form

Syntax Tree



Quadruple

	Op	Arg1	Arg2	RESULT
(0)	=	a		T1
(1)	if	T1	2	8
(2)	-	c	d	T2
(3)	if	T2	4	6
(4)	+	a	c	T3
(5)	goto	9		
(6)	*	a	c	T4
(7)	goto	9		
(8)	+	a	b	T5
(9)	goto	10		
(10)	... next statement ...			

Triple

	Op	Arg1	Arg2	
(0)	=	a		
(1)	if	(0)	3	(Condition TRUE)
(2)	goto	10		(Condition FALSE)
(3)	-	c	d	
(4)	if	(3)	6	
(5)	goto	8		
(6)	+	a	c	
(7)	goto	12		
(8)	*	a	c	
(9)	goto	12		
(10)	+	a	b	
(11)	goto	12		
(12)	... next statement ...			

Fig 2.4 Comparison of each type of intermediate code

A statement such as A := E + 10 was generated as

```
T1 = 10
T2 = E + T1
A  = T2
```

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.

	Op	Arg1	Arg2	Rsl
(2)	=	10		T1
(1)	+	E	T1	T2
(2)	=	T2		A

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.

```
A  ->  id := E
E  ->  [E+]E | [E-]E
F  ->  [F*]C
C  ->  id | int | (E)
```

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 := B * (C + D)

is shown in fig 2.7.

INPUT	STACK	PLACE	GENERATED CODE
A := B*(C+D)			
:= B*(C+D)	id	A	
B*(C+D)	id :=	A_	
*(C+D)	id := id	A_B	
*(C+D)	id := O	A_B	T1 = B
*(C+D)	id := F	A_T1	
(C+D)	id := F*	A_T1_	
C+D)	id := F*(	A_T1_	
+D)	id := F*(id	A_T1_ _C	
+D)	id := F*(O	A_T1_ C	
D)	id := F*(F	A_T1_ C	
+D)	id := F*(F	A_T1_ _C	
D)	id := F*(F+	A_T1_ C	
)	id := F*(F+id	A_T1_ _C_D	
)	id := F*(F+C	A_T1_ _C_D	
)	id := F*(F+F	A_T1_ _C_D	T2 = C + D
)	id := F*(F	A_T1_ _T2	
	id := F*(F)	A_T1_ T2_	
	id := F*O	A_T1_ _T2_	T3 = T1 * T2
	id := F	A_T3	A = T3
	A	A	

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 := B*(C+D). The field PLACE 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,

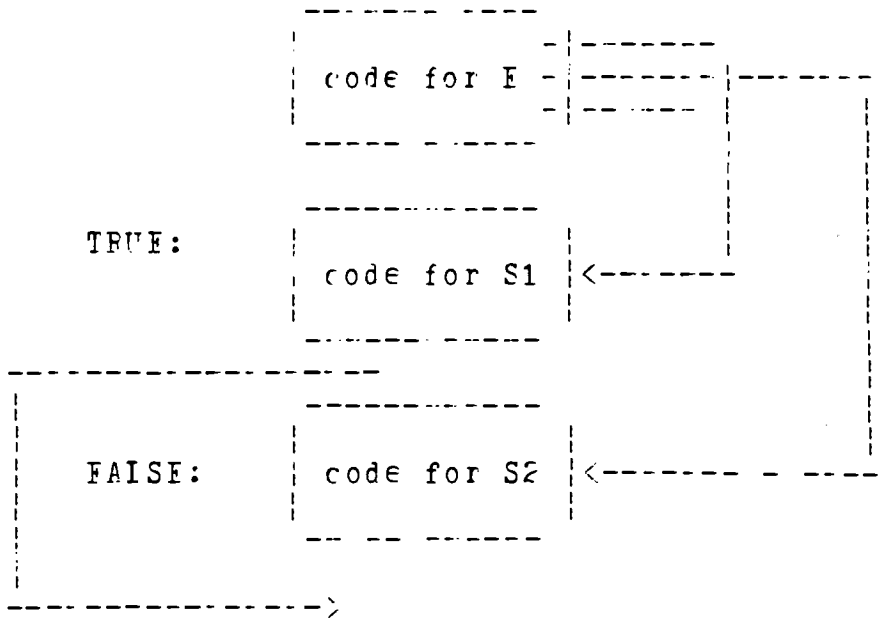
if a then if c-d then a c else a*c else a+b

The a and c-d are supposed to be boolean expressions. Both of them will be translated into a sequence of quadruples. Fig 2.8 gives a general idea of translation for conditional statements such as

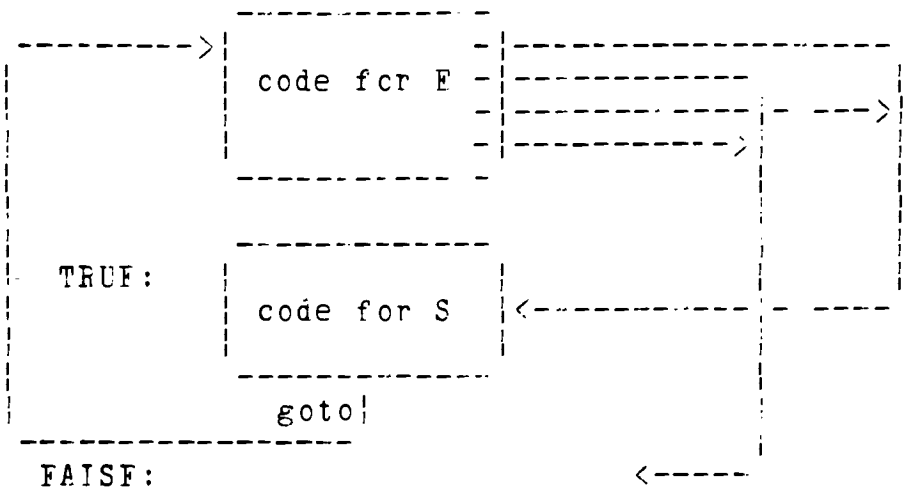
if E then S1 else S2

and

while E do S



(a) if-statement



b) while-statement

Fig 2.9 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 branching 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 when 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 gains 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 2.9 shows how to perform an interpretation algorithm.

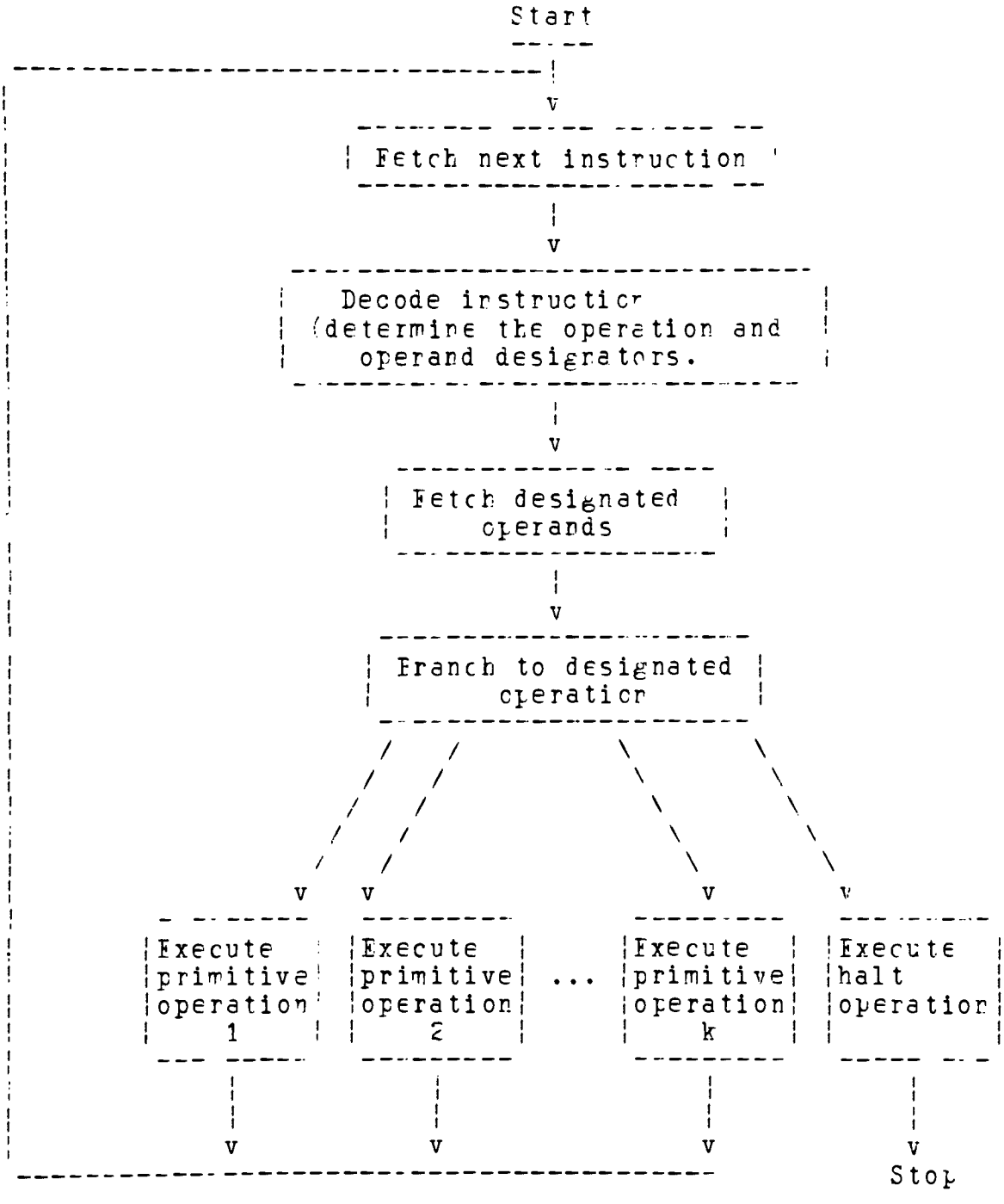


Fig 2.9 Basic procedure for program interpretation and execution.

The "xeq" function does whatever each quadruple tells it to do. After checking the "Op 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.

Op	QUADRUPLE Arg1	Arg2	Rsl	"xeq" (interpretation)	Note
<	A	B	T1	T1 = 1 if A < B, T1 = 0 otherwise.	All comparison (< > !=) has the same idea.
[]	T1	T2	T3	T3 = T1 [T2	Case ITEM in "xeq". (Mini-language Type
if	T1	3	7	-if T1 = 1 (true) set i to 3, 7 if it is not.	
call	0	3	-	-save the index of the next quad in a "calladdr" stack. -transfer control to the called procedure by assigning 0 to i.	
param	A	-	-	-save the value of A by pushing it on the "quev" stack.	
que	W	-	-	-assign W to the value taken from the bottom of "quev" stack.	
parama	A	-	-	-save the address of A by pushing it on the "quea" stack.	

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 out 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 A in the form "Rsl A = 5". if A has the value of 5.

Fig 2.10 list of key quadruples and the result of interpretation

CHAPTER 3

MINI-LANGUAGES

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

3.1 MINI-LANGUAGE CORE

Each Miri-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 B.

A semicolon is used as a termination symtol 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 "Op Rsl Arg1 Arg2" and any effect in symbol table.

3.1.3 Declaration

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

Example: declare A, F, 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: READ(identifier)
[READ(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,

"data" input	input statement	result

```
0    5                  input A, B, C;          A := 0
16                                              B := 5
                                                C := 16
```

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, B, C;

Quadruple Format: WRITE(identifier)
[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,

Output Statement	Action
output A, B, C;	Rsl A = 0 Rsl B = 5 Rsl C = 16

The output starts on a new line.

3.1.7 If Statement

Format: if comparison then statement-sequence
[else statement-sequence]
end if;

Example:
```
if (A < B) then
  A := A + B;
  B := B - A;
else   A := A - B;
end if;
```

Quadruple Format:
IF(compare result, true target, false target)

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

BACKPATCH(p,i)

means "make each of the quadruples on 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, corsider the nested-if statement shown in Fig 3.2

```
if (A < B) then
     C := A + B;
     if (B < C) then
      C := B;
     else
      C := A;
     end if;
else
    if (A    B) 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.

Statement	Quadruple generated				
		Op	Arg1	Arg2	Rsl
if (A < B) then	100:	<	A	B	T1
	101:	if	T1	102	_
C := A + B;	102:		A	B	T2
	103:	=	T2		C
if (B < C) then	104:	<	B	C	T2
	105:	if	T3	106	_
C := B;	106:	=	B		C
else	107:	goto	_		
	Fix the Rsl field in 105 with 108.				
C := A;	108:	=	A		C
end if;	109:	goto	_		
else	Fix the Rsl field in 101 with 110.				
if (A = B) then	110:	=	A	B	T4
	111:	if	T4	112	

A := C;	112: = C A
end if;	113: goto 114 Fix the Rsl field in 111 with 114.
C := A + B;	114: + A B C
end if;	115: goto 116 Fix the Arg1 field in 107 and the Arg1 field in 109 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.

	Op	Arg1	Arg2	Rsl
100:	<	A	B	T1
101:	if	T1	102	110
102:	+	A	B	T2
103:	=	T2		C
104:	<	B	C	T3
105:	if	T3	106	108
106:	=	B		C
107:	goto	116		
108:	=	A		C
109:	goto	116		
110:	=	A	B	T4
111:	if	T4	112	114
112:	=	C		A
113:	goto	114		
114:	+	A	B	C
115:	goto	116		
116:	:			
117:	:			

Fig 3.4 Intermediate code for Mini-language Core.

As mentioned, the quadruple format for the if statement is

IF(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

Format:
```
while comparison loop
statement-sequence
end loop;
```

Example:
```
while (WEIGHT ` 150  loop
 IOSS := IAST IOSS ' 5;
 IAST_LOSS := IOSS;
 WEIGHT := WEIGHT - IOSS;
end loop;
```

Quadruple Format:
IF'compare result, true target, false target)

Symtol 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.

```
 98:  :
 99:  :
100:  >      WEIGHT     150      T1
101:  if     T1         102      107
102:  =      5                   T2
103:  +      IAST_IOSS  T2       IOSS
104:  =      IOSS                IAST_IOSS
105:  +      WEIGHT     IOSS     IAST_IOSS
106:  goto   100
107:     :
108:     :
```

Fig 3.5 Intermediate code for while statement.

3.1.9 Comparisons

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

Example: (SMALL < BIG)

Quadruple Format:
IT GT
EQ NEQ(first value,second value, target value)

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
MUL(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

s1 s2 ... sn

of two or more D-structures S1 through sn.

* Conditional structure

```
if   c    ther
     s1
else
     s2
end if;
```

when c is a condition and s1 and s2 are D-structures.

* Iterative structures

```
while    c   loop
         s
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.

```
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.

  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. Consequencely, 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, fig 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 Boehm and Jacopim [1966] in D-structures. He simply explained the proof of the theorem (1E'. 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 program 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, 'PAD' and '4B5'), 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 D. There are four tables; Identifier table, Variable table, Type table and Store table.

(1) Identifier table : 'idtab' 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 'typeador' 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 : 'typetab' with four fields included.

'Sort'

'field1'

'field2'

'field3'

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).

chain : 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

ARRAY	size	chain	type

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

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

type : type of elements (Type row number).

SSRANGE : subscript range

SSRANGE	str	chain	type

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.

INTSUB : integer subrange

STRSUB : string subrange

BOSUB : boolean subrange

INTSUB STRSUB BOSUB	field1	lb	ub

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

lb : Indicates the lower bound of the array.

ut : 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[i1,i2,. .,ik] where ith index runs from lower bound i (IBi) up to some di

for each dimension i=1,2,....k. Mathematically, we give this element A[i1,i2,...,ik] offset

$$(i1-LB1)D2D3....Dk + (i2-LB2)D3D4...Dk +...+ (i(k-1)-LB(k-1))Dk + (ik-LBk) \qquad (T.1)$$

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

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

Dj can be calculated in advance in the lexical analyzer phase. Finally, the three-address code for i, for example, A[i1,i2,i3,i4] is

```
T1 = i1 - LB1
T2 = T2 * D1
T3 = i2 - IB2
T4 = T3 * D2
T5 = T2 + T4
T6 = i3 - IB3
T7 = T6 * D2
T8 = T5 + T7
T9 = i4 - IB4
T10 = T9 * D3
T11 = T8 + T10
T12 = 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.FIRSTNAME 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 IICENSE structure is started (base address of the structure). At this point the 'type' index is used to specify where 'IICENSE' 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 subname 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 (IICENSE) A + B). The answer will be the index in 'store' table pointing to the value, IICENSE.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 techriques 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 (lower

bound) and 100 (upper bound). For example, one may have:

```
declare X,Y,GCOD;     -- 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

procedure.

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-

eter.

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

- PAR (parameter)
- PROCEDURE (name of procedure)
- SIMPLE (integer in main program)
- ARRAY (array)

(3) Store : Pointer to the other three tables, depends on Type field.

Possible Type	Point to
PAR	'partab' (parameter table)
PROCEDURE	'proctab' (procedure table)
SIMPLE	'storetab' (store table)
ARRAY	'storetab' (store 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) Quard# : 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' (parameter table).

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) Ptype : The mode of the parameter (Value, Result, Value-result, Location)

(4) Pstore : 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 compactity. The purpose of Henry Ledgard 1981 [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-language 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.

BIBLIOGRAPHY

(1) "Reference Manual for the Ada Programming Language", U.S. Department of Defense, July 1980 (Also published by Spring-Verlag, NewYork, Spring 1981).

(2) Aho Alfred V., The Theory of Parsing. Translation and Compilation Vol.I : Compiling , Prentice-Hall Englewood Cliffs, New Jersey.

(3) Aho Alfred V. and Ullman Jeffery D., Principles of Compiler Design, Bell Laboratories Murray Hill, New Jersey, Addison-Wesley Publishing company (1978)

(4) Barrett William A. and Couch John D., Compiler Construction : Theory and Practice. Science Research Associations Inc.(1979),

(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

Conference 112-18 (1981).

(6) Gannon John D. and Horning J.J., "Language Design for Programming Reliability", IEEE Transactions of Software and Engineering, Vol.1, No.2, June 1975.

(7) Goodenough J.B., "The Ada Compiler Validation Capability". Sigplan Not. (USA) Vol.15, No 11, 1-9 November 1980.

(8) Goos G. and Wintersteir G., "Towards a compiler front-end for Ada", Sigplan Not.(USA) Vol.15, No.11, 36-46 November 1980.

(9) Gries David, Cornell University and Narain Gehani, The State University of New York at Buffalo, 'Some Ideas on Data Types in High-level Languages". Communications of the ACM, June 1977, Vol.20. No 6.

(10) Hisger Andy, Iamt David Alex, Rosemberg Johnathan, and Sherman Mark, Carnegie-Mellon University, 'A Runtime Representation for Ada Variables and Types", Sigplan Notices Vol.15, No.11, November 1980.

(11) Hopcroft John E. and Ullman Jefferey D., Introduction to Automata Theory, Languages, and Computation, Addison-Wesley Publishing Company (1979).

(12) Johnson Stephen C., Yacc - Yet Another Compiler-Compiler", Bell Laboratories, Murray Hill, New Jersey 07974 (July 31,1979).

(13) Ledgard Henry F., "Ada, An Introduction, Ada Reference Manual", (July 1980), Spring-Verleg NewYork Heidelberg Berlin.

(14) Ledgard Henry F., "Ten Mini-Languages : A Study of Topical Issues in Programming Languages , Computing Surveys, Vol 3, No.3, September 1971.

(15) Ledgard Henry F., and Marcotty Michael, The programming Language Landscape, The SRA Computer Science Series (1981).

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

(17) Iewis II P.M., Rosenkrantz D.J., and Stearns R.E., Compiler Design Theory Addison-Wesley Publishing Company (1978)

(18) Mills Harlen D., "Mathematical Theory of Computation" IBM Corporation Report FSC 71-6012, Gaithersburg Maryland, February 1972

(19) Pratt Terrence W. Programming Languages : Design and Implementation Prentice-Hall Inc. Englewood Clifts, N.J.(1975)

(20) Pyle I.C. The Ada Programming Language Prentice Hall International (1981)

(21) Richard Frederic and Ledgard Henry F. "A reminder of Language Designs", Sigplan Notices, December 1977

(22) Rosenberg Jonathan, Lamb David Alex, Hisgen Andy, and Sherman Mark (Carnegie-Mellor University) "The Charrette Ada Compiler", Sigplan Notices, Vol.15, No.11, November 1980

(23) Wetherell C.S. 'Problem with the Ada Reference Grammar", Sigplar Notices. (USA) Vol.16, No. 9, 90-104, September 1981

APPENDICES

6.1 APPENDIX A

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

Context Free Syntax of Miri-language in BNF

Some extensions to BNF were used to described the grammars of Miri-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 BNF 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,

variable ::= identifier
| identifier [expression]

Table F.1 Context Free Syntax of Mini-language Core in BNF

```
<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> )

<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 F.2 Context Free Syntax of Mini-language L in BNF

```
<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 <condition-expression> then
                                     <statement-sequence>
                                 [ else
                                     <statement-sequence>   ]
                                   end if;

<loop-statement>             ::=   while <condition-expression  loop
                                     <statement-sequence>
                                   end loop;

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

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

<condition-expression>       ::=   [ <condition> and ] <condition>
                                   | <condition> or  ] <condition>

<condition>                  ::=   <comparison>
                                   ( <condition-expression> )

<comparison>                 ::=   ( <operand><comparison-operator <operand>

<integer-expression>         ::=   [ <operand  + ] <operand
```

```
                    |     [ <operand> - ] <operand>

<operand>           ::=   <integer>
                    |     <identifier>
                    |       <integer-expression> )

<comparison-operator ::=  < | = | ! | >

<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 E.3 Context Free Syntax of Mini-language Type in BNF

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

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

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

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

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

<type>                       ::=   <simple-type>
                                 | <array-type>
                                 | <record-type>

<simple-type>                ::=   <integer>
                                 | <string>
                                 | <boolean>

<array-type>                 ::=   array [ <bounds> ] of <type>

<record-type>                ::=   record
                                     <identifier> : <type> ;
                                   [ <identifier> : <type> ; ]...
                                   end record;

<bounds>                     ::=   <integer>..<integer>

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

<assignment-statement>       ::=   <variable> := <expression> ;

<if-statement>               ::=   if <expression> then
                                     <statement-sequence>
                                   [ else
                                     <statement-sequence> ]
                                   end if;

<input-statement>            ::=   input <variable> [ , <variable> ]... ;
```

```
output-statement>        ::=   output <variable> [ , 'variable> ]... ;

<expression>             ::=  [ <operand> 'operator> ] <operand

<operand>                ::=   'variable>
                          |    <integer>
                          |    <string
                          |    .toolean>
                          |  [ 'expression> ]

<string>                 ::=   ' <character>. .'

<toolean>                ::=   true
                          |    false

<operator>               ::=   < | = | != | > |   | - | * | /
                          |    cat | ard | or

'character>              ::=   <letter>
                          |    <digit>
                          |    <special-character

<special-character>      ::=   (sp) | + | - | * | ' | : | ; |
                          |    . | . | $ | %   = | != | > | <

<letter>                 ::=   A | B | C | D | E | 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
```

Note: (sp) means space or blank character.

Table E.4 Context Free Syntax of Mini-language Procedures in BNF

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

<variable-declaration>     ::=   <declaration>
                           |     <declaration> <variable-declaration>

<procedure-declaration     ::=   <procedure>
                           |     <procedure> <procedure-declaration>

<procedure>                ::=   procedure <identifier> <parameter-list>
                                    <variable-declaration>
                                 begin
                                    <statement-sequence>
                                 end ;

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

<parameter>                ::=   <identifier> : <parameter-mode>

<parameter-mode>           ::=   value
                           |     result
                           |     value-result
                           |     location
                           |     name

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

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

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

<statement>                ::=   <assignment-statement>
                           |     <input-statement
                           |     <output-statement>
                           |     <call-statement>

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

<call-statement>           ::=   identifier <expression-sequence
```

```
<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 | 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
```

6.2 APPENDIX F

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.

ExampleC.1 for A Mini-language 'Core'

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

Rsl  COUNT    = 2
Rsl  NEXT_TERM    = 3
Rsl  LAST_TERM    = 2
Rsl  THIS_TERM    = 3
Finish xeq

Data: 2
```

ExampleC.2 for a Mini-language 'Core'

```
line :
 1     feclare CCUNT **syntax error**, LIMIT;
**syntax error: line 1: Missing program key word

 2       declare LAST_TERM, THIS_TERM, NEXT_TERM;
 3    begin
 4       CCUNT  0 **syntax error**;
 5       LAST_TERM := 1;
 6       THIS_TERM := 1;
 7       input LIMIT;
 8
 9       ( **syntax error**CCUNT < LIMIT)
 10          output LAST_TERM;
 11          NEXT_TERM := LAST_TERM + THIS_TERM;
 12          LAST_TERM := THIS_TERM;
 13          THIS_TERM := NEXT_TERM:
 14          COUNT := COUNT + 1;
 15       end loop;
 16    end
 17
number of errors 3
```

ExampleC.3 for a Mini-language 'Core'

```
line :
 1 program
 2     declare B,T,TT;
 3   begin
 4     input B,T,TT;
 5     output B,T,TT;
 6     if (B < 3) then
 7       while (T > 2) loop
 8         T := T + 1000001;
 9         TT :=  TT   B;
10       end loop;
11     else
12       B := 4;
13     end if;
14     if (B !=  5) then
15       B := 4;
16     endif;
17     output B,T,TT;
18   end;
19
Start xeq

Rsl  B    = 2
Rsl  T    = 3
Rsl  TT    - 0
*** Variable out of range ( ) 555555555 or - 0) ***


Data: 2 3
```

ExampleC.4 for Mini-language 'Core'

```
line :
 1 program
 2      declare P,TT,D.H.T,J,G,I,K,P,A,F;
 3    begin
 4        while (T    F) loop
 5          TT := TT - 1;
 6        end loop;
 7        if (T < T) then
 8          P := D;
 9        end if;
10         while (T!=G) loop
11            if (I=J) then
12              if (Y=P) 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=P) then
25           E := U;
Undefined identifier: U

26         end if;
27    end;
28
number of errors 1
```

ExampleD.1 for a Mini-language 'D'

```
line :
 1 program
 2      --  This program reads in an integer value representing the time
 3      --  on a 24-hour clock and prints out the corresponding 12-hour
 4      --  clock time. If the input value does not represent a correct
 5      --  time, the input value is printed.
 6
 7      declare TIME, HOURS_AND_MINUTES, HOURS, MINUTES, AM_OR_PM;
 8    begin
 9      input TIME;
10       HOURS_AND_MINUTES := TIME;
11       HOURS             := 0;
12       while (HOURS_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
18       if (HOURS > 23) then
19           if (HOURS = 24) and (MINUTES = 0) then
20               AM_OR_PM := 0;
21               HOURS    := 12;
22               output HOURS, MINUTES, AM_OR_PM;
23           else
24               output TIME;
25           end if;
26       else
27           if (MINUTES > 59) then
28               output TIME;
29           else
30               AM_OR_PM := 0;
31               if (HOURS = 0) then
32                   HOURS := 12;
33               else
34                   if (HOURS > 11) then
35                       AM_OR_PM := 1;
36                       if (HOURS > 12) then
37                           HOURS := HOURS - 12;
38                       end if;
39                   end if;
40               end if;
41               output HOURS, MINUTES, AM_OR_PM;
42           end if;
43         end if;
44       end;
45
46
```

```
Start xeq

Rsl HOURS    = 2
Rsl MINUTES  = 32
Rsl AM_OR_PM   1

Data: 1432
```

ExampleP.1 for Mini-language 'Procedure'

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

Rsl  I    = 3
Rsl  A[3]   = 6
Rsl  I    = 3
Rsl  A[3]   - 6
Finish xeq
```

ExamleP.2 for Miri-language 'Procedure'

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

Rsl  I    = 3
Rsl  A[3]    = 6
Rsl  I    =
** Error: Attempt to evaluate ar undefired variable
Rsl  A[3]    =
** Error: Attempt to evaluate an undefined variable
Finish xeq
```

ExampleP.3 for a Mini-language 'Procedure'

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

Rsl  I    = 3
Rsl  A[3]   = 6
Rsl  I    = 6
Rsl  A[3]   = 3
Finish xeq
```

ExampleP.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 := 5;
 9       V := CC + C;
10      end;
11     procedure PP(G : value  :
12      declare II;
13      begin
14       II := 2 * G;
15       output II;
16      end;
17   begin
18     input A,S;
19     A := A + S;
20     output A,S;
21     P(1,A,S);
22     output A,S;
23     PP(A);
24   end;
25
Start xeq

Rsl  A    = 4
Rsl  S    = 3
Rsl  A    = 6
Rsl  S    = 3
Rsl  II   = 8
Finish xeq
```

ExampleT.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];
 8             A[3][2][6] := '8+2' cat 'rat';
 9             A[4][2][6] := 'good ;
10              output A[3][2][6];
11              output A[3][3][5];
12    end;
13
start xeq
Rsl A[3][2][6]    '8+2rat'
Rsl A[3][3][5]    'CC'
Finish xeq


Data: 'CC'
```

ExampleT.2 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                  integer;
 6   begin
 7             input A[3][2][4];
 8             input A[3][3][5];
 9             AA[3][2][6] := A[3][2][4] + 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
```

ExampleT.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];
 10             A[3][6][4] := A[3][5][4] and A[3][5][4] or 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
```

Example1.4 for a Mini-language 'Type'

```
line :
 1 program
 2   declare A,B: record Y : integer;
 3                       Z :  record
 4                              B : integer;
 5                              U : record
 6                                  G : boolean;
 7                                  GG : string;
 8                                  end record;
 9                              BB : 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.B;
 19            output A.Y;
 20            A.Y := 4;
 21    A.Y := A.Z.B   A.Y;
 22            output A.Y, A.Z.B, A.Z.U.G;
 23   end;
 24
start xeq
Rsl A.Y    6
Rsl A.Y    12
Rsl A.Z.B    8
Rsl A.Z.U.G   false
Finish xeq


Data : false
       6
       8
```

ExampleI.5 for a Mini-language 'Type'

```
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 : 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.B < A.C.CC.DDD) then
18             output A.C.CC.DDF;
19             end if;
20             A.B := 4;
21             input A.E;
22             A.E := A.B cat A.E;
23             A.E := 'yes' cat 'nop';
24             A.C.CC.DDD := A.B * 12;
25             output A.B, 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
Rsl A.B    4
Rsl A.C.CC.DDD    48
Rsl A.E    'yesnop'
Rsl A.C.CC.DDF
Finish xeq
```

6.3 APPENDIX C

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 in fig C.1, should be considered.

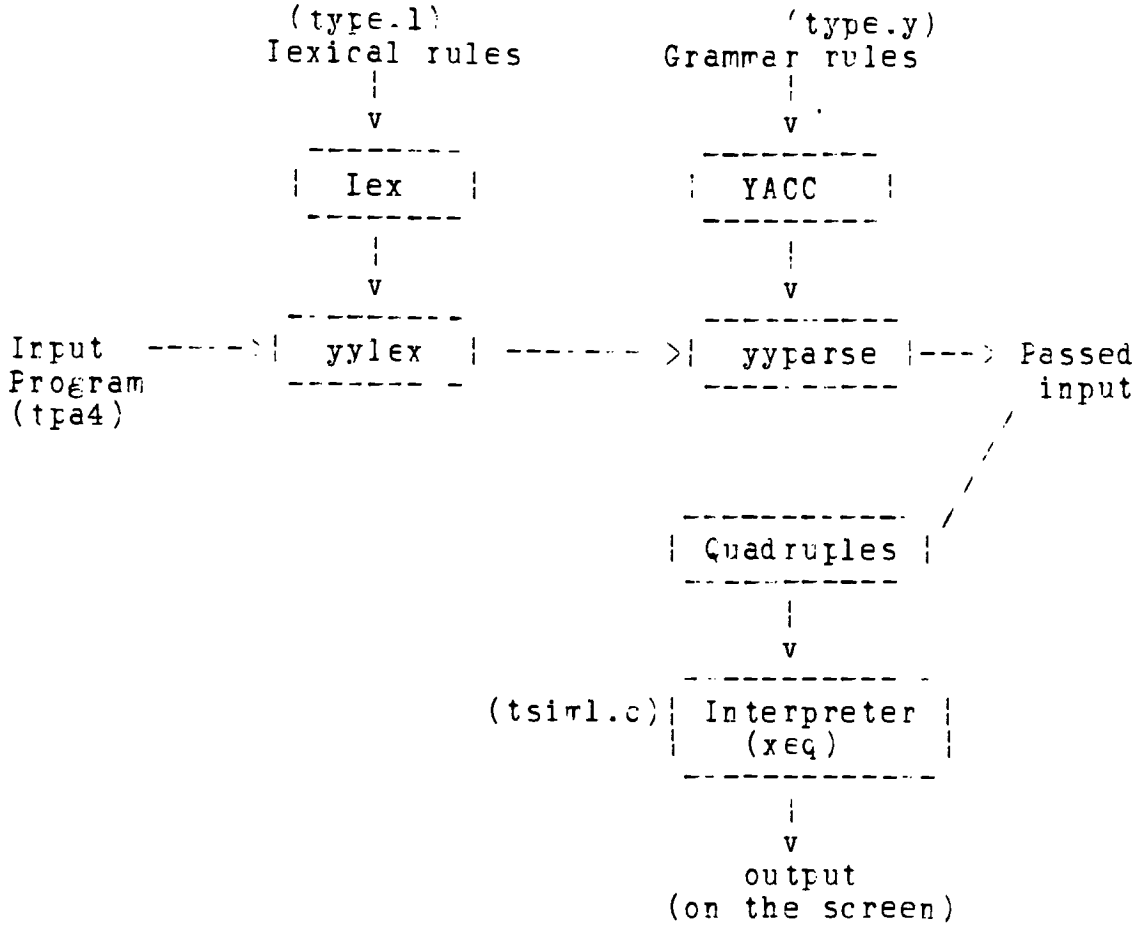


Fig C.1 Structure used in each implemented Mini-language (in this thesis).

Example Mini-language Type (record type)

File name : tpaf

```
program
declare A,X,Y :
            record
              F : integer;
              E : string;
              C : record
                  CC : record
                       DDL : integer;
                       DDE : 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.E := 'yes' cat 'nop';
        A.C.CC.DDD := A.F * 12;
        output A.F, A.C.CC.DDD, A.E;
        A.C.CC.DDE := A.C.CC.DDE cat A.C.CC.DDE;
        output A.C.CC.DDE;
end;
```

Cutput listing after the abcve example was executed.

Command : a.out < trae

```
line :
 1 program
 2   declare A,X,Y :
 3               record
 4                 B : integer;
 5                 E : string;
 6                 C : record
 7                     CC : reccrd
 8                          DDD : integer;
 9                          DDE : string;
10                           DDF : boclean;
11                           end record;
12                      RC : boolean;
13                      end reccrd;
14                erd record;
15   begin
16             A.C.CC.DDF := true and true;
17             if(A.B < A.C.CC.DDD) then
18             output A.C.CC.DDF;
19             end if;
20             A.B := 4;
21             A.E := 'yes' cat 'nop';
22             A.C.CC.DDD := A.B * 12;
23             output A B, A.C.CC.DDD, A.E;
24             A.C.CC.DDE := A.C.CC.DDE cat A.C.CC.DDF;
25             output A.C.CC.DDE;
26   end;
27
start xeq
Rsl A.B   =   4
Rsl A.C.CC.DDD   -   48
Rsl A.E   =   'yesnop'
Rsl A.C.CC.DDE   =
Finish xeq
```

Symbol tables

File name : idtab.code

1.) id table (idtab)
---

	name	varaddr
0	A	0
1	X	1
2	Y	2
3	T1	3
4	T2	4
5	T3	5
6	T4	6
7	T5	7
8	T6	8
9	T7	9
10	T8	10
11	T9	11
12	T10	12
13	T11	13

File name : var.code

2.) variable table (vartab)
---

	typeaddr	storeaddr
0	17	0
1	17	6
2	17	12
3	18	18
4	19	19
5	20	20
6	21	21
7	22	22
8	23	23
9	24	24
10	25	25
11	26	26
12	27	27
13	28	28

File name : type.code

3.) type table (typetab)

	sort	field1	field2	field3
1	RECFID	66	3	2
2	INTSUB	0	0	00000000
3	RECFID	69	5	4
4	STRSUB	1	1	1
5	RECFID	67	0	16
6	RECFID	773	14	13
7	RECFID	622	9	8
8	INTSUB	2	0	00000000
9	RECFID	623	11	10
10	STRSUB	3	1	1
11	RECFID	624	0	12
12	BOSUB	4	1	1
13	RECORD	3	7	0
14	RECFID	332	0	15
15	BOSUB	5	1	1
16	RECORD	2	6	0
17	RECORD	3	1	0
18	BOOLEAN	1	1	1
19	BOOLEAN	1	1	1
20	BOOLEAN	1	0	00000000
21	BOOLEAN	1	0	00000000
22	INTEGER	1	0	00000000
23	STRING	1	1	1
24	STRING	1	1	1
25	STRING	1	0	0
26	INTEGER	1	0	00000000
27	INTEGER	1	0	00000000
28	STRING	1	0	0

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.B  A.C.CC.DDD
8  if  T4  6
 write  A.C.CC.DDF
8  goto
T5  =  4
A.B  =  T5
T6  =  'yes'
T7  =  'nop'
T8  cat  T6  T7
A.F  =  T8
T9  =  12
T10  *  A.B  T9
A.C.CC.DDD  =  T10
 write  A.B
 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
```

Jun 30 13:14 1983 pns0578 tmain.c

```
/*****************************************************************/
/* This program is the main program of a compiler                */
/* for Mini-language Type. It dose                               */
/*       - 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 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"


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


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

 /* begining 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 out information, if want to *
        dumpgen();
        dumpsym();
        dumpidtab();
        dumptype();
        dumpvar();

     /* Print the errors, if any; continue the interpreter, if not */
        if (nmerrs> 0)  printf(" number of errors %d\0",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("\0");
         }
}


/* print out my own error massage */

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


/*********************/
/* yacc error handling */
```

```
yyerror (s)
char *s;
{
        extern int en, numline,nmerrs;
        lnerr = 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 debug the programs.               */

makename(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 RECFID:    return (strcpy(str,"RECFID"));
        case RECSUB:    return (strcpy(str,"RECSUB"));
        case STRING:    return (strcpy(str,"STRING"));
        case BOOLEAN:   return (strcpy(str,"BOOLEAN"));
        case INTSUB:    return (strcpy(str,"INTSUB"));
        case STRSUB:    return (strcpy(str,"STRSUB"));
        case BOSUB:     return (strcpy(str,"BOSUB"));
        case SSRANGE:   return (strcpy(str,"SSRANGE"));
        case GT:        return (strcpy(str,">"));
        case EQ:        return (strcpy(str,"="));
        case NEQ:       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);
        fprintf(obj,"    sort        field1      field2        field3  \0);
        for(i=1;i<numtype;i++)
        {
          fprintf(obj,"%d  %s   ",i,(makerme(typetab[i]->sort)));
          fprintf(obj,"       %d   ",typetab[i]->field1);
          fprintf(obj,"       %d   ",typetab[i]->field2);
          fprintf(obj,"       %d  \0,typetab[i]->field3);
        }
        fclose(obj);
}


/* dump "var" table */

dumpvar()
{
        FILE *obj,*fopen();
        struct symvar *prt;
        int i;
        extern numvar;
        obj = fopen("var.code", "w");
        fprintf(obj," variable table (vartab)  ----------- ");
        fprintf(obj,"\0ypeaddr    storeaddr");
        for(i=0;i<numvar;i++)
        {
         fprintf(obj,"117d %d   ",i,vartab[i]->typeaddr);
         fprintf(obj,"          %d   ",vartab[i]->storeaddr);
        }
        fprintf(obj,"\0");
        fclose(obj);
}


/* dump "id" table */

dumpidtab()
{
        FILE *obj,*fopen();
        struct nlist *ptr;
        int i;
        obj = fopen("idtab.code","w");
        fprintf(obj," id table (idtab)  -- --- -");
        fprintf(obj," name     varaddr");
        for(i=0;i<ck;i++)
        {
          fprintf(obj,"117d  %s   ",i,keep[i]->name);
          fprintf(obj,"       %d   ",keep[i]->varaddr);
```

```
        }
        fprintf(obj,"0);
        fclose(obj);
}


/* 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<scount;i++)
              {
               fprintf(obj,"%d  ",i);
               prt = storetab[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 printf(" bad code in dumpsym i %d",i);
                    }
                      }
               fprintf(obj,"0);
              }
            fclose(obj);
}
```

Jun 30 13:14 1983 pns2578 type.l

```
%{
#include "y.tab.h"
#include "th.h"
#include "tdec.h"
YYSTYPE yylval;
%}

LETTER          [A-Z]+
DIGIT           [0-9]+
BLANK           [ ]+
NL              [\n]
%%
                        extern list, dec;
                        int flag = 's';
                        extern index ;
                        extern numline;
                        extern firstarray. array, arraysize;
program                 {
                           ECHO;
                           list = norm;
                           return PROGRAM;
                        }
begin                   {
                           ECHO;
                           return START;
                         }
declare                  {
                           ECHO;
                           dec = true;
                           list = iddec;
                           return DECLARE;
                        }
if                      {
                           ECHO;
                           switch (flag)
                           {
                             case 's':
                                      return IF;
                             case 'e':
                                      flag = 's';
                                      return ENDIF;
                           }
                           ECHO;
                        }
then                    {
                           ECHO;
                           return THEN;
                        }
```

```
else                    {
                            ECHO;
                            return ELSE;
                        }
record                  {
                            ECHO;
                            switch (flag)
                            {
                              case 's':   firstarray = true;
                                          return RECORD;
                              case 'e': flag = 's'; return ENDREC;
                            }
                            ECHO;
                        }
end                     {
                            flag = 'e';
                            ECHO;
                        }

input                   {
                            ECHO;
                            list = readin;
                            return INPUT;
                        }
output                  {
                            ECHO;
                            list = writeout;
                            return OUTPUT;
                        }
array                   {
                            ECHO;   array   true;
                            return ARRAY;
                        }
of                      {
                            ECHO;
                            return OF;
                        }
true                    {
                            ECHO;
                            return TR;
                        }
false                   {  ECHO; return FS;  }
integer                 {  ECHO;
                           return INTEGER; }
string                  {  ECHO; return STRING;  }
boolean                 {  ECHO; return BOOLEAN; }
":"                     {  ECHO;
                           return COLON;  }
"["                     {  ECHO; index = true; return LBK;    }
"]"                     {  ECHO; index = false; return LBK;
"."                     {  ECHO; return DOT;    }
```

```
","                     {
                            ECHO;
                            return MORE;
                        }
';'                     {
                            ECHO;
                            switch (flag)
                            {
                              case 's':
                              return FS;
                              case 'e':  flag = 's';
                                         return ENDPROG;
                            }
                        }
":="                    {
                            ECHO;
                            list = asgn;
                            return ASSIGN;
                        }
'('                     {
                            ECHO;
                            return OPEN;
                        }
")"                     {
                            ECHO;
                            return CLOSE;
                        }
'<'                     {
                            ECHO;
                            return LT;
                        }
">"                     {
                            ECHO;
                            return GT;
                        }
"="                     {
                            ECHO;
                            return EQ;
                        }
!=                      {
                            ECHO;
                            return NEQ;
                        }
"+"                     {
                            ECHO;
                            return ADD;
                        }
'-'                     {
                            ECHO;
                            return MINUS;
                        }
```

```
"*"                         {
                               ECHO;
                               return MUI;
                            }
'/"                         {  ECHO; return DIV;  }
cat                         {  ECHO; return CAT;  }
and                         {  ECHO; return AND;  }
cr                          {  ECHO; return CR;   }
                               strcpy(yylval.id,yytext);
                               return STROP; }
{LETTER}(("_"{LETTER})?)*   {
                               ECHO;
                               strcpy(yylval.id,yytext);
                               return IDENTIFIER;
                            }
{DIGIT}                     {
                               ECHO;
                               sscanf(yytext,"%d",&yylval.num);
                               return INTCP;
                            }
{NI}          |
--.*{NI}                    {
                               ECHO;
                               numline++;
                               printf("  %d     ,numline 1);
                            }

%%
```

Jun 30 13:14 1983 pns2578 type.y

```
%{
#include "tquard.h"
#include "thh.h"
#include "th.h"
#include "tdec.h"
#include "tsym.h"
#include <stdio.h>
#include "dftype.h"
#include "type.h"

%}

%start program

%token      PROGRAM  START ENDPROG
            DECLARE  IF  THEN
            INPUT OUTPUT ENDIF ELSE
            MORE ASSIGN NL BLANK
%token      INTEGER IDENTIFIER
%token      FS ADD MINUS MUL
            EQ NEQ LT GT OPEN CLOSE
%token      INTCP STRCP STRING BOOLEAN
            DIV CAT AND OR COLON DOT ARRAY
            CPK CFK OF RECORD ENDREC TR FS
            INDEX


%right        ASSIGN
%left         EQ NEQ
%left         LT GT
%left         ADD MINUS
%left         MUL  DIV


%%      /* beginning of the program */

program         :     prog
                      therestofprog
                      {
                       gen(STOP,NULL,NULL,NULL);
                      ;

proc            :     PROGRAM
                      {
                       for(i=0;i<MAXID;i++)
                       {
                        wcsve[i] = 0;
```

```
                    D[i] = 1;
                  }
                }

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

therestofprog   :       declaration
                        content
        ;

declaration     :       dec
                        idenlist COICN
                        {}
                        type
                        ES
                        {
                         DC = 1;
                         if($5.num == ARRAY)
                          $$.tp = puttype(ARRAY,(arraysize=poparrsize()),
                                          poparrfst(),typearray);
                         for(i=0;i<idcount;i++)  {
                         switch ($5.num) {
                          case INTEGER:
                                $$.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],$$.vp[i]);
                                break;
                          case BOOLEAN:
                                $$.tp = puttype(BOOLEAN,where++,1,1);
                                storetab[scount] = intstore(true);
                                $$.vp[i] = putvar($$.tp,scount++);
                                install(idsave[i],$$.vp[i]);
                                break;
                          case RECORD:
                                $$.vp[i] = putvar($5.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" ;
                         break;
                               } /* case statement */
                                  } /* for statement */
                }
        ;

dec             :       DECLARE

                |       error ES
                        { usage("Missing declare keywcrd");  }
                ;

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();
                          BOUNDS = 1;
                        }

                |       recordtype
                        { $$.num = RECORD;
```

```
                    firstarray = true;
                  }
              ;

simpletype    :     INTEGER
                    {
                      $$.num = INTEGER;
                    }

              |     STRING
                    {
                      $$.num = STRING;
                    }

              |     BOOLEAN
                    {
                      $$.num = BOOLEAN;
                    }
              ;

arraytype     :     ARRAY CBK bounds CBK OF
                    {
                    pushfix(ARRAY);
                    arraycount--;
                    typetmp = puttype(SUBRANGE,$3.num.0,0);
                    if(firstarray) {   pusharrfst(typetmp);
                                       firstarray = false;
                                   }
                    else    { BOOLARR = true;
                              typetab[poparr()]->field2 = typetmp;
                            }
                    pusharr(typetmp);
                    pusharr(typetmp);
                    }
                    type
                    {
                    extern arrstk[20], arrptr;
                    if(notfix) {  poparr(); notfix = false;  }
                    TYPE = $7.num;
                    while (morearray)  {
                     switch(TYPE)  {
                     case INTEGER:   typearray = INTEGER;
                          $$.tp = puttype(INTSUB,INTEGER,poplb(),popub(
                          typetab[poparr()]->field3   $$.tp;
                          morearray = false;      break;
                     case STRING:    typearray = STRING;
                          $$.tp = puttype(STRSUB,STRING,poplb(),popub()
                          typetab[poparr()]->field3 = $$.tp;
                          morearray = false;      break;
                     case BOOLEAN:   typearray = BOOLEAN;
                          $$.tp = puttype(BOOSUB,BOOLEAN,poplb(),popub()
```

```
                        typetab[tmp=poparr()]->field3 = $$.tp;
                        morearray = false;      break;
                    case ARRAY:     TYPE = typearray;
                        break;
                    case RECORD:    typearray = RECORD;
                        arraystore = arraystore / BOUNDS;
                        $$.tp = puttype(RECSUB,$7.tp,poplb(),popub()
                        typetab[poparr()]->field3 = $$.tp;
                        morearray = false;
                        break;
                                }    /* case */
                                }   /* while */
                    if(changearrsize)
                     pusharrsize(arraysize);
                    changearrsize = false;
                    arraysize = 1;
                    firstarray = true;
                    morearray = true;
                    }
                ;

bounds          :    INTOP DOT DOT INTOP
                     {
                       pushlb($1.num);
                       pushub($4.num);
                       $$.num = $4.num - $1.num +1;
                       BOUNDS = $$.num;
                       arraysize = $$.num * arraysize;
                       arraystore = $$.num * arraystore;
                       lb[DC] = $1.num;
                       ub[DC] = $4.num;
                       ++DC;
                       for(i=1;i<DC;i++)
                        D[i] = BOUNDS * D[i];
                       changearrsize = true;
                     }
                ;
recordtype      :    reckey
                     {
                      if(wc > 0 && typeptr > -1)
                      {
                        wcsve[wc-1] = wordcount;
                        popsave[rc++] = poptype();
                      }
                      temp   wc+ ;
                      temp = temp + nub;
                      wordcount = wcsve[temp];
                     }
                     recseq
                     ENDREC
                     {
```

```
                    else firstrec[++fr] = numtype-1;
                    pushtype(numtype-1);    pushtype(numtype-1);
                    }
                    CCION
                    {}
                    type ES
                    {
                     switch ($5.num) {
                     case INTEGER:
                          $$.tp = puttype(INTSUB,where++,MININT,MAXINT);
                          putstore($5.num);
                          break;
                     case BOOLEAN:
                          $$.tp = puttype(BOSUB,where++,1,1);
                          putstore($5.num);
                          break;
                     case STRING:
                          $$.tp = puttype(STRSUB,where++,1,1);
                          putstore($5.num);
                          break;
                     case RECORD: break;
                     case ARRAY:
                          $$.tp = puttype(ARRAY,(BOUNDS=poparrsize()),
                                  poparrfst(),typearray);
                          where = where + BOUNDS +1;
                          break;
                     default: printf(" bad type in rec0);
                              break;
                                    }
                     if(typeptr > -1 && $5.num != RECORD)
                     {
                      tmp = poptype();
                      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 'begin' key word"); }
                ;

endprog         :   ENDPROG
```

```
                |   error
                    { usage("Missing end");  }
                ;

statseq         :   statseq  stat

                |   stat
                ;

stat            :   ifstat

                |   nonifstat
                ;

nonifstat       :   assignstat

                |   inputstat

                |   outputstat
                ;

assignstat      :   variable ASSIGN express ES
                    {
                      $$.slen = (char*)malloc(strlen($1.slen) 1);
                      strcpy($$.slen,$1.slen);
                      switch(typetab[$1.tp]->sort) {
                      case INTEGER: case INTSUB:
                           if(typetab[$3.tp]->sort != INTEGER &&
                              typetab[$3.tp]->sort != INTSUB)
                           {  printf(" Type combination");
                              nmerrs++; }
                           else
                             gen(ASSIGN,strelem($3.slen),NULL,
                                 strelem($$.slen));
                           break;
                      case STRING: case STRSUB:
                           if(typetab[$3.tp]->sort !  STRING &&
                              $3.tp != STRING &&
                              $3.tp != STRSUB &&
                              typetab[$3.tp]->sort != STRSUB)
                           { printf(" Type combination");
                             nmerrs++; }
                            else
                              gen(ASSIGN,strelem($3.slen),
                                  NULL,strelem($$.slen));
                            break;
                       case BOOLEAN: case BOSUB:
                            if(typetab[$3.tp]->sort != BOOLEAN &&
                               typetab[$3.tp]->sort != BOSUB)
                            {  printf(" Type combination");
                               nmerrs;
```

```
                        }
                        else
                          gen(ASSIGN,strelem($3.slen),NULL,
                          strelem($$.slen));
                        break;
                   default: printf(' wrong opcode type');
                                                  }
                  list = norm;
                }

        !       variable error ES
                  {  usage('Assign operator expected ':=' ;  }
        ;


ifstat          :       firstif therestofif

        ;

firstif         :       IF express THEN
                  {
                     int i;
                     elseflag[0] = true;
                     if(elseflag[ii] == true){ ii++;
                              elseflag[ii] = false;
                              endifflag[ii] = false;
                                              }
                     push(addr);
                     gen(IF,strelem($2.slen),intelem(addr 1),NULL);
                     ++ifcount[ii];
                  }
        ;

therestofif     :       endifpart

        !       statseq ELSE
                  {
                     int tmpp;
                     int inttmp;
                     elseflag[ii] = true;
                     if(endifflag[ii] == true)
                     tmpp = pop();
                     fixup(intelem(addr+1),pop(),4);
                     if(endifflag[ii] == true)
                     push(tmpp);
                     push(addr);
                     gen(GOTO,NULL,NULL,strelem(NULL));
                  }
                endifpart
        !       error ES { usage('Forgot 'end if;' keyword');  }
        ;
```

```
endifpart       :     statseq endkey
                ;
endkey          :     ENDIF ES
                       {
                          int inttemp, j, i;
                          if(--ifcount[ii] == 0) {
                            /* go to higher level */
                            gen(GOTO,NULL,NULL,
                                 (intelem(inttemp-addr 1)));
                            if(elseflag[ii] == true)  {
                               /* fix else */
                             elseflag[ii] = false;
                             if(maxstk >0)
                               fixup(intelem(inttemp),pop(),4);
                                                       }
                            if(maxstk>0)
                             /* fix if */
                             fixup(intelem(inttemp),pop(),4);
                             ii--;
                             if(ii == 0 && maxstk > 0)
                               /* clear if stack at end */
                              while(maxstk > 0)
                               fixup(intelem(inttemp),pop(),4);
                                                       }
                          else {
                                 if(elseflag[ii] == true) {
                                     elseflag[ii] = false;
                                    if(maxstk > 0)
                                     fixup(intelem(addr),pop(),4);
                                                       }
                                 else fixup(intelem(addr),pop(),4);
                                 if(endifflag[ii] = true) {
                                   endifflag[ii] = false;
                                   if(maxstk>0)
                                    fixup(intelem(addr),pop(),4);
                                                       }
                                 push(addr);
                                 gen(GOTO,NULL,NULL,strelem(NULL));
                               }
                          endifflag[ii] = true;
                       }

                |      error ES
                       {
                          usage("Forgot 'end if;' keyword");
                       }

                ;

inputstat       :      INPUT varlist ES
                       {   list = norm; }
```

```
                INPUT varlist error ES
                {
                    usage("Missing ';'");
                }
        ;

outputstat      :   OUTPUT
                    varlist ES
                    {  list = norm;  }

                |   OUTPUT varlist error ES
                    {
                       usage('Missing ';'");
                    }
        ;

varlist         :   varlist MORE variable
                    {
                      $$.slen = (char*)malloc(strlen($3.slen) 1);
                      strcpy($$.slen,$3.slen);
                      switch(listsave)  {
                      case readin:
                           gen(IN,strelem(iosave),NULL,NULL);
                           break;
                      case writeout:
                           gen(OUT,strelem(iosave),NULL,NULL);
                           break;
                      default: break;
                                       }
                    }

                |   variable
                    {
                      $$.slen = (char*)malloc(strlen($1.slen)+1);
                      strcpy($$.slen,$1.slen);
                      switch(listsave)     {
                      case readin:
                           gen(IN,strelem(iosave),NULL,NULL);
                           break;
                      case writeout:
                           gen(OUT,strelem(iosave),NULL,NULL);
                           break;
                      default: break;
                                           }
                    }
        ;

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(BOOLEAN,1.MININT,MAXINT);
                 storetab[scount++] = intstore(true);
                 $$.place = scount - 1;
                 install($$.slen,putvar($$.tp,$$.place) ;
                }
                if($1.num == CAT)
                {
                 $$.tp = puttype(STRING,1.0,0);
                 storetab[scount++] = strstore(NULL);
                 $$.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++] = intstore(0);
                 $$.place = scount - 1;
                 install($$.slen,putvar($$.tp,$$.place) ;
                }
               }

        |      operand
               { $$ = $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++] = intstore(0);
                  $$.place = scount - 1;
```

```
                    gen(MUI,strelem(strtemp),intelem(I[DC]),
                        strelem(strtempM));
                    DC++;
                    if(notfirst)  {
                     strtemp = newtemp();
                     $$.tp = puttype(INTEGER,1,MININT,MAXINT);
                     storetab[scount++] = intstore(0);
                     $$.place = scount -1;
                     install(strtemp,putvar($$.tp,$$.place));
                     gen(ADD,strelem(strtmpsv),strelem(strtempM),
                         strelem(strtemp));
                     strtmpsv = strtemp;
                                 }
                    else  strtmpsv = strtempM;
                    notfirst = true;
                    $$.slen = strtmpsv;
                           }  /* PK */
                   else   {
                   list = norm;
                   strtemp = newtemp();
                   gen(ASSIGN,intelem($1.num),NULL,strelem(strtemp
                   $$.slen = strtemp;
                   $$.tp = puttype(INTEGER,1,MININT,MAXINT);
                   storetab[scount++] = intstore($1.num);
                   $$.place = scount - 1;
                   install($$.slen,putvar($$.tp,$$.place));
                           }
                   fig = $1.num;
                }

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

        |       OPEN express CLOSE
                {
                 $$ = $2;
                }
        ;

operator        :       IT  { $$.num = IT;  }

        |       EQ  { $$.num = EQ;  }
```

```
        |   NEC { $$.num = NEQ; }
        |   GT  { $$.num = GT;  }
        |   ADT { $$.num = ADD; }
        |   MINUS {  $$.num = MINUS;  }
        |   MUI { $$.num = MUL; }
        |   DIV { $$.num = DIV; }
        |   CAT { $$.num = CAT; }
        |   AND { $$.num = AND; }
        |   OR  { $$.num = OR;  }
        ;

variable    :   variabletmp
                {
                  idfirst = true;
                  numplace = 0;
                  switch($1.num)  {
                  case RECORD:
                       $$.place = get();
                       if(notfound)  {
                         printf(" Undefined variable );
                         nmerrs++;
                         return(NULL);
                                       }
                       $$.tp = savetype;
                       $$.slen = (char*)malloc(strlen($1.slen)+1);
                       strcpy($$.slen,$1.slen);
                       iosave = $$.slen;
                       break;
                  case ARRAY:
                       strtmpsv = strtemp;
                       strtemp = newtemp();
                       $$.tp = puttype(INTEGER,1,MININT,MAXINT);
                       store tab[scount++] = intstore(555555);
                       $$.place = scount -1;
                       install(strtemp,putvar($$.tp,$$.place));
                       getflag = true;
                       tmp = get();
                       getflag = false;
                       gen(ASSIGN.intelem(tmp),strelem(NULL),
                            strelem(strtemp) ;
                       gen(ARRAY,strelem(strtemp),strelem(strtmpsv),
                            strelem(strtemp)=newtemp() );
                       if(listsave==writecut || listsave== readin)
```

```
                {
                 iosave = (char*)malloc(strlen($1.slen) +1);
                 strcpy(iosave,$1.slen);
                }
                $$.slen = strtempM;
                $$.tp= puttype(ITEM,1,MININT,MAXINT);
                stcretat[scount++] = intstore(0);
                $$.place = scount -1;
                install($$.slen,putvar($$.tp,$$.place));
                $$.tp=typetat[typetat[$1.tp]->field2]->field3
                break;
        default:
                $$.slen = (char*)malloc(strlen($1.slen)+1);
                strcpy($$.slen,$1.slen);
                iosave = $$.slen;
                break;
                        }  /* case */
        testfirst = true;
        cin - 0;
        DC = 1;
        itemrcount=0;
        notfirst = false;


    ;

variabletmp     :   variabletmp DOT IDENTIFIER
                {
                 if(testfirst) {
                  varsave[cin] = (char*)malloc(strlen($1.slen)+1);
                  strcpy(varsave[cin++],$1.slen);
                  testfirst = false;
                                }
                 varsave[cin] = (char*)malloc(strlen($3.id)+1);
                 strcpy(varsave[cin++],$3.id);
                 $$.slen = (char*)malloc(strlen($1.slen)+
                           strlen($3.id)+2);
                 sprintf($$.slen,"%s.%s ",$1.slen,$3.id);
                 $$.num = RECORD;
                }

                variabletmp  CHK
                { HK = true; }
                express CHK
                {
                  HK = false;
                  if(listsave==writeout ||
                     listsave==readin)              {
                  $$.slen = (char*)malloc(strlen($1.slen) 5);
                  sprintf($$.slen,"%s[%d] ",$1.slen,fie);
                                                    }
```

```
                    else $$.slen = $4.slen;
                    idtemp = newtemp();
                    $$.tp = puttype(INTEGER,1,MININT,MAXINT );
                    storetab[scount++ ] = intstore(555555);
                    $$.place = scount -1;
                    $$.num = ARRAY;
                    if(fig < lb[DC-2] || fig > ub[DC-2])
                    {
                     printf(" ** Index out of bound");
                     nmerrs++ ;
                    }
                    $$.tp = savetype;
                  }

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

toclean       :   TR
                  {  $$.num = true;  }
              |   FS
                  {  $$.num = false; }
              ;

%%
```

Jun 30 13:14 1983 pns0578 y.tab.h

```
# define PROGRAM 257
# define START 258
# define ENDPROG 259
# define DECLARE 260
# define IF 261
# define THEN 262
# define INPUT 263
# define OUTPUT 264
# define ENDIF 265
# define ELSE 266
# define MORE 267
# define ASSIGN 268
# define NL 269
# define BLANK 270
# define INTEGER 271
# define IDENTIFIER 272
# define ES 273
# define ADD 274
# define MINUS 275
# define MUL 276
# define EQ 277
# define NEQ 278
# define LT 279
# define GT 280
# define OPEN 281
# define CLOSE 282
# define INTOP 283
# define STROP 284
# define STRING 285
# define BOOLEAN 286
# define DIV 287
# define CAT 288
# define AND 289
# define CR 290
# define COLON 291
# define DOT 292
# define ARRAY 293
# define OPK 294
# define CPK 295
# define OF 296
# define RECORD 297
# define ENDREC 298
# define TR 299
# define FS 300
# define INDEX 301
```

Jun 30 13:14 1983 prs2578 tstk.c

```
/*****************************************************************/
/* This file contains all the two-stacks functions used in */
/* in Mini-Language Type compiler (mostly, in type.y).     */
/*****************************************************************/


#define MAXTYPE 30

int typestk[MAXTYPE],  typeptr=0,  maxtype=0;
int arrstk[MAXTYPE],   arrptr=0,   maxarr=0;
int ubstk[MAXTYPE],    ubptr=0,    maxub=0;
int lbstk[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++ > 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++ > MAXTYPE)  printf(" over flow type stack");
        typestk[typeptr++] = val;
```

```
        return;
}


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

/*****************************************************************************/

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

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

/*****************************************************************************/

pushlb(val)
int val;
{     if(maxlb++ > MAXTYPE)   printf(" over flow lb stack");
      lbstk[lbptr++] = val;    return;          }

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

/*****************************************************************************/

pushub(val)
int val;
{    if(maxub++   MAXTYPE)   printf(" over flow ub stack");
```

```
    utstk[utptr++] = val;     return;          }

poput()
{    int inttemp;     irttemp = --ubptr;       maxut- ;
     return(utstk[inttemp]);
}

/*****************************************************************************/

pusharrfst(val)
int val;
{       if(maxaf++ > MAXTYPE)  printf(" over flow arrfst stack" );
        afstk[afptr++] = val;  return;          }

poparrfst()
{       int inttemp;   inttemp = --afptr;       maxaf--;
        return(afstk[inttemp]);
}

/*****************************************************************************/

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

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

/*****************************************************************************/
```

Jun 30 13:14 1983 prs0578 tsym.c

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

#define RECFIL 960
#define true 1
#define false 0
#define NULL 0
#define MAXVAL 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 *lookup(s        /* look for s in symbol table */
char *s;
{
        struct nlist *np;
        for (np = idtab[hash s)]; np != NULL; np = np->next)
           if (strcmp(s,np->name)==0)
                 return(np);              /* found it */
        return (NULL);                    /* not found */
}
```

```
/*******************/
int trace = 0;
struct nlist *install(name,usage) /* put in hash table */
char *name;
int usage;
{
        struct nlist *np, *lookup();
        char *strsave(), *malloc();
        int *intsave();
        int hashval;

        if ((np - lookup(name)) == NULL)  {       /* not found *
          if(list == asgn || list == readin || list == writeout){
                nmerrs ;
                printf(" Undefined identifier: %s 0,name);
                return (NULL);
                        }
          np = (struct nlist *)malloc(sizeof(*np) ;
          if (np == NULL)
               {
                  printf(" np");
                  return (NULL);
               }
          if ((np->name - strsave(name))== NULL)
               {
                  printf(" strsave ");
                  return (NULL);

               }
          np->varaddr - usage;
          hashval   hash(np->name);  /* produces starting index in */
          np->next = idtab[hashval];     /* array hashtab */
          idtab[hashval] = np;
          keep[ck++] - 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 0;
        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 alloctuf[AIICCSIZE];
static char *allocp = alloctuf;

char *nalloc(n)       /* for 'strsave' function */
int n;
{
        if(allocp + n <= alloctuf + AIIOCSIZE) {
           allocp += n;
           return(allocp - n);
        } else
           {
             return(NUII);
           }
}
```

```
**********/
/* Put the information of the variable in type table ("typetab") */

int puttype(sort,field1.field2,field3)
int sort, field1;
int field2;
int field3;
{
        extern numtype;
        struct symtype *new;
        new = (struct symtype*)malloc(sizeof(struct symtype));
        new->sort = sort;
        new->field1 = field1;
        new->field2 = field2;
        new->field3 = field3;
        typetab[numtype++] = new;
        dumptype();
        if(numtype > MAXTYPE)
         printf(" Type space is exhausted.Expand MAXTYPE in tsym.h");
        return (numtype-1);
}


/**********/
/* Also put the information in variable table ("vartab") */

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);
}
```

Jun 30 13:14 1983 pns2578 tstore.c

```
/*****************************************************************/
/* This file contains functions that will put in a               */
/* value in symtol tables; vartat, storetat, typetat.            */
/*****************************************************************/


#irclude "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 jj;

int i, ij;


/*****************************************************************/

struct symstore *intstore(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;                   exterr typearray;
        extern arraystore;                  exterr array;
        int BOUNDS;
        extern struct symstore *intstore();
        extern struct symstore *strstore();
        BOUNDS = arraystore;
        if(first == false)
         for(k=0;k<idcount;k++)    {
          switch (type)  {
          case INTEGER:
               for(ii=0;ii<BOUNDS;ii++)
               storetab[scount++] = intstore(0);
               break;
          case STRING:
               for(ii=0;ii<BOUNDS;ii++)
               storetab[scount++] = strstore(NULL);
               break;
          case BOOLEAN:
               for(ii=0;ii<BOUNDS;ii++)
               storetab[scount++] = intstore(true);
               break;
          default: break;
                          }  /* case */
          firststore[i] = i;
          first = true;
                                  }
        else      {
          jj = 0;
          isave = (scount-1) / idcount;
          iisave = iisave + BOUNDS;
          while(jj++ < idcount-1, {
           for(i=scount-1;i>isave;i--)
             storetab[i+BOUNDS] - storetab[i];
           switch (type) {
           case INTEGER:
                for(ii=0;ii<BOUNDS;ii++)
                storetab[ii+i+1] - intstore(0);
                break;
           case STRING:
                for(ii=0;ii<BOUNDS;ii++)
                storetab[ii+i-1] = strstore(NULL);
                break;
           case BOOLEAN:
                for(ii=0;ii<BOUNDS;ii++)
                storetab[ii i 1]   intstore(true);
                break;
           default: break;
```

```
                        } /* case */
            firststore[jj] = i+2;
            isave = i+1 iisave;
            scount = scount + BOUNDS;
                                    }    /* while jj*/
           /* fix the last one */
           switch (type) {
           case INTEGER:
                for(ij=0;ij<FOUNDS;ij++)
                storetab[scount++] = intstore(0`; ;  break;
           case STRING:
                for(ij=0;ij<FOUNDS;ij++)
                storetab[scount+ ] = strstore(NULL);
                break;
           case BOOLEAN:
                for(ij=0;ij<FOUNDS;ij++)
                storetab[scount++] = intstore(true,;
                break;
           default: printf(" bad putstore`; break;
                        }  /* case2 */
              }    /* else */
        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 -] = intstore(true);   break;
         defalt:   printf(" wrong arrstore  );
                        }
         if(i == 0)  arrfst = scount-1;
                                    }
}
```

Jun 30 13:14 1983 pms0578 get.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 fuction gives the address  */
/* (index to 'storetab' array) variable.          */

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

        mp = vartab[idtab[hash(varsave[0])]->vareddr];
        keepplace = mp->storeeddr;
        if(getflag)
        {
           getflag = false;
           return(keepplace);
        }
        nnp = typetab[mp->typeaddr];
        if(nnp->sort == INTEGER || nnp->sort == STRING ||
           nnp->sort == BOOLEAN || nnp->sort==ITEM ||
           nnp->sort==INTSUB || nnp->sort==STRSUB) {
           savetype = mp->typeaddr;
           numplace - 1;
           return (keepplace);
                                }
        else {
            notfound - true;
            vc = 0;
            keepplace   getplace(nnp,mp);
            if(vc != cin-1)  notfound = true;
```

```
            else
             return (keepplace);
             }
}


struct symvar *mmp;

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

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

        switch (nnp->sort)  {
        case INTEGER:    case STRING:    case BOOLEAN:
        case ITEM:  /* array case */
             return (keepplace);
        case INTSUB:     case STRSUB:    case BOSUB:
             sub = (nnp->field1);
             keepplace = (mp->storeaddr)+sub;
             return (keepplace);
        case RECFLD:
             numplace= ;
             mmp = mp;
             if(nnp->field1 == hash(varsave[++vc])) {
              notfound = false;
              if(typetab[nnp->field3]->sort == INTSUB ||
                 typetab[nnp->field3]->sort == STRSUB ||
                 typetab[nnp->field3]->sort == BOSUB
               savetype = nnp->field3;
              if((keepplace=getplace(typetab[nnp->field3],mp)) == NULL)
                   return (NULL);
              else
                   return(keepplace);
                                                        }
             else
                {
                 notfound = true;
                 --vc;
                 if((keepplace=getplace(typetab[nnp- field2],mp))== NULL)
                   return(NULL);
                }
             break;
        case ARRAY:
```

```
        savetype    typetab[nrp->field2]->field3;
        break;
case RECORD:
        mmp = mp;
        notfound = true;
        for(i=0;i<nrp->field1 && notfound;i++ )
            keepplace = getplace(typetab[nrp->field2],mp);
        if(keepplace == NULL) return (NULL);
        else
              return(keepplace);
        break;
default:
        printf(" wrong sort ");
        printf(" nrp->sort %d",nrp->sort);
        break;
                        }
```

Jun 30 13:14 1983 prs2578 tgen.c

```
#include "y.tab.h"
#include "thh.h"
#include "tquard.h"
#include "tsym.h"
#include "tdec.h"
#include <stdio.h>


/* import variables */
extern struct symquard *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 aquard *quardsave[MAXQUARD];


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

gen(op,arg1,arg2,rsl)
int op;
struct quard *arg1,*arg2,*rsl;
{
        struct aquard *new;    /* create temporary area */
        new = (struct aquard*) malloc(sizeof(struct aquard));
        new->Op    = op;
        new->Arg1 = arg1;
        new->Arg2 = arg2;
        new->Rsl   = 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 'quard.h  0);
        return;
}


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

struct quard *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,"T%d ",tempcount);
        return(str);
}


/*********************/
/* function for generating a temporary name "T1...Tn" */

char *newindtmp()
{
        char *str;
        itempcount++;
        str = (char*)malloc(IDMAX 1);
        sprintf(str,"I%d ",itempcount);
        return(str);
}
```

```
/*********************/
/* finishing 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]->Rs1  = 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 out off stack   */

int pop()
{
        int inttemp;
        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 quardruples.                         */

dumpgen()
{
        FILE *obj, *fopen();
        int i 0;
        struct quard *prt;
        char *a - " ";
        obj = fopen("gq.code","w");
        while(i<addr)
        {
            prt = quardsave[i]- Rsl;
               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,0);
               i++;
        }
        fclose(obj);
}


/*********************/
/* This fuction 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 OUT:       return (strcpy(str,"write" ));
case STOP:      return (strcpy(str,"stop" ));
case ARRAY:     return (strcpy(str,"[]" ));
default:        return (strcpy(str,"bad" ));
```

Jun 30 13:14 1983 pns0579 tsim1.c

```
#include <stdio.h>
#include "tsym.h"
#include "tdec.h"
#include "y.tab.h"
#include "thh.h"
#include "dftype.h"
#include "tquard.h"
#include "tsim.h"
#define MAXID 20
#define HASHSIZE 787

/* import variables and structures */
extern struct symquard *symint;
extern struct nlist *symtab[HASHSIZE];
extern struct nlist *install();
extern struct symquard *value;
extern addr;
extern struct aquard *quardsave[MAXQUARD];
extern int in[10];
extern numvar;
char *varsave[MAXID];
char *ITEMA, *ITEME;

struct symstore *AA;
int cin, A, F, errors=0, iq=0;
int item = false;
int arg1place, arg2place, rslplace;
int rsltype, arg1type, arg2type;

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

       /* temporary variables used here */
          int typesave[MAXTYPE], tt, count=0, k=0, j, temp;
          char *T, *F, *aa, *string;
       /***/

      /* used in GROUP */
        extern get();
        extern getvar();
        extern savetype;
        extern struct symtype *typetab[MAXTYPE];
        extern getflag;
     /***/

        getflag = false;
```

```
T = "true";   F - "false ;

/* start from the first quardruple */

 while(iq<addr)           {
   temp = quardsave[iq]- Cp;
   switch (temp`                                {

   case ASSIGN :  GROUP3;
        temp = typetab[rsltype]->sort;
        switch(typetab[rsltype]->sort)   {
        case STRING: case STRSUP:
             if(*ARG1ID == ''')  /* case 'string' */
              strsim(rslplace,ARG1ID);
             else {
                       GROUP1;
                       strsim(rslplace,STARG1ID);
                  }
             break;
        case INTEGER: case INTSUP: case BOOLEAN: case BOSUP:
             if(ARG1TP==Int)  intsim(rslplace,ARG1NUM);
             else {    GROUP1;
                       intsim(rslplace,STARG1NM);
                  }
             break;
        case ITEM:
             GROUP1;
             if(SORT1==ITEM)
             {
              GROUP3;
              if(ITEMETP= Str)
                strsim(STRSINM,ITEM1ID);
              else
              intsim(STRSINM,ITEM1NM);
             }  /* SORT1   ITEM */
             else
             {
              if(SORT1==STRING || SORT1-=STRSUB)
               strsim(STRSINM,STARG1ID);
              else
               intsim(STRSINM,STARG1NM);
             }
             break;
        default: printf("bad type in ASSIGN");
                                                  }
        item = false;
        break;
   case ARRAY:
        item = true;
        math();
        STRSITP = Int;
```

```
     STRSINM = A + B;
     break;
case ADD :
     math();      STRSITP = Int;     STRSINM = A + B;
     break;
case MINUS:
     math();      STRSITP = Int;     STRSINM = A - B;    break;
case MUL:
     math();      STRSITP = Int;     STRSINM = A * B;    break;
case DIV:
     math();      STRSITP = Int;     STRSINM = A / B;    break;
case CAT:
     GROUP1;   GROUP2;   GROUP3;
     if('SCRT1==STRING || SCRT1==STRSUB || SCRT1== ITEM) &&
        'SCRT2==STRING || SORT2==STRSUB || SOPT2==ITEM))
      {
        if(SCRT1==ITEM) ITEMA = ITEM1ID;
        else ITEMA = STARG1ID;
        if(SCRT2==ITEM);      ITEMB = ITEM2ID;
        else ITEMB = STARG2ID;

        /* test if both are null string */

        if(*ITEMA != ''  && *ITEMB !  ' ') ;
        else {
        string = (char*)malloc(strlen(ITEMA)
                               strlen(ITEMB)+1);
        STRSIID = string;
        sprintf(string, "%s%s ".ITEMA.ITEMB);
        mystrcpy(STRSIID,string);
        STRSITP = Str;
              }
      }
     else usage("variable type string required");
     break;
case LT :  GROUP1;  GROUP2;  GROUP3;
     GETOPCODE();
     if(A < B) intsim(rslplace,true);
     else intsim(rslplace,false);      break;
case GT :  GROUP1;  GROUP2;  GROUP3;
     GETOPCODE();
     if(A > B) intsim(rslplace,true);
     else intsim(rslplace,false);      break;
case EQ :  GROUP1;  GROUP2;  GROUP3;
     GETOPCODE();
     if(A == B) intsim(rslplace,true);
     else  intsim(rslplace,false);     break;
case NEQ : GROUP1;  GROUP2;  GROUP3;
     GETOPCODE();
     if(A != B) intsim rslplace.true);
     else intsim(rslplace,false);      break;
```

```
case OR :  GROUP1;  GROUP2;  GROUP3;
     GETOPCODE();
     if(A == false && B == false) intsim(rslplace,false);
     else   intsim(rslplace,true);     break;
case AND : GROUP1;  GROUP2;  GROUP3;
     GETOPCODE();
     if(A == true && B == true) intsim(rslplace,true);
     else   intsim(rslplace,false);     break;
case IF :
     GROUP1;
     if(STARG1NM == true) iq = quardsave[iq]->Arg2->term.numval-
     if(STARG1NM == false) iq = quardsave[iq]->Rsl- term.numval-
     break;
case GOTO :
     iq = quardsave[iq]->Rsl-.term.numval-1;
     break;

case IN :
     GROUP1;
     if(item) {
      B = SCRT1;
      iq = iq -1;
      GROUP3;
      A = STRSINM;
      iq = iq + 1;
               }
     else B   typetab[arg1type]  sort;
     typesave[count++] = B;
     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]==INTSUB)
       fscanf(obj,"%d",&bb);
      else /* it is a boolean or string type */
        fscanf(obj,"%s",aa);
     }
     if(typesave[count-1]== INTEGER || typesave[count-1]== INTSUB
        /* data type is an integer */
       if(item)
          fscanf(obj,"%d",&(ITEMRSINM));
       else fscanf(obj,"%d",&(STARG1NM));
     else {     /* data type is a string or boolean *

          fscanf(obj,"%s",aa);
          if(typesave[count-1]= STRING
          || typesave[count-1]== STRSUB)
             /*data type is a string */
            if(test(aa) == true
                if(item) {
                 ITEMRSIID = (char*)malloc(strlen(aa)+1);
```

```
                strcpy.ITEMRSIID,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,F)==0) {
                if(item) ITEMRSINM = false;
                else STARG1NM = false;
                                    }
                    else {    errors++;
                              printf("  *** Illegal data ***");
                              break;
                         }
               }
          }
     fclose(obj);
     item = false;
     break;
case OUT :
     if(errors == 0){
     printf(" Rsl %s   -   ",quardsave[iq]->Arg1->term.idval);
     GROUP1;
     if(item) {
      B = SORT1;
      iq = iq - 1;
      GROUP2;
      AA = storetab[STRSINM];
      iq = iq + 1;
             }
     else {
      AA = storetab[arg1place];
      B = typetab[arg1type]->sort;
          }
     if(B == BOOLEAN || B == BOSUP)
      if(AA->ustore.storenum == 1) printf("true");
      else printf("false");
     else xeqprint(AA);
                    }
     item = false;
     break;

case STOP :
     printf(" Finish xeq 0);
     return;
default : printf("  Can not find opcode );
```

```
                                        }
          iq+ ;
                                    }
}


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

xeqprint(prt)
struct symstore *prt;
{
        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 data *** "; errors+ ;
```

```
        return (NULL);
    }
    while(*s != ' ') s--; /* scan untill the last character */
    if(*(s-1) != '"') {
      printf(" *** illigal data ***"); errors++;
      return (NULL);
                      }
    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 != ' ')
    {
     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++],s);
     return;
}


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

math()
{
              if(ARG1TP == Str) {
                   GROUP1;
                if(SCPT1==ITEM)
```

```
        {
          A    storetat[STARG1NM]->ustore.storenum;
        }
        else
            A = STARG1NM;
                          }
      else A = ARG1NUM;
      if'ARG2TP == Str  {
            GROUP2;
        if(SORT2==ITEM)
        {
          B    storetat[STARG2NM]->ustore.storenum;
        }
        else
            B = STARG2NM;
                          }
      else B = ARG2NUM;
      GROUP3;


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

GETOPCODE()
{
              /** check at argument1  **/
               if(SORT1==ITEM)
                if(ITEM1TP != Int) {  PROMPSS;   errors++ ;  }
                else A = ITEM1NM;
               else
                if'STARG1TP != Int) { PROMPSS;   errors++ ;  }
                else A = STARG1NM;

              /** look at the argument2 **/
               if(SORT2==ITEM)
                if(ITEM2TP != Int)  { PROMPSS;    errors++;  }
                else B = ITEM2NM;
               else
                if(STARG2TP != Int) { PROMPSS;    errors++;  }
                else  B = STARG2NM;



/*******************/
intsim(val1,val2)
int val1;
int val2;
{
        storetat[val1] >storetype = Int;
```

```
        storetab[val1]->ustore.storenum = val2;
}

strsim(val1,val2)
int val1;
char *val2;
{
        storetab[val1]->storetype = Str;
        storetab[val1]->ustore.storeid = (char*)malloc(strlen(val2)+1);
        strcpy(storetab[val1]->ustore.storeid,val2);
}
#define Int 900
#define Str 910
#define MAXID 20
#define HASHSIZE 787

struct symstore {
        int storetype;
        union {
        int storenum;
        char *storeid;
              } ustore;
                 } ;
struct symstore *storetab[HASHSIZE];
```

Jun 30 13:14 1983 pns2578 tstore.h

```
#define Int 900
#define Str 910
#define MAXID 20
#define HASHSIZE 787

struct symstore {
        int storetype;
        union {
        int storenum;
        char *storeid;
              } ustore;
                 } ;
struct symstore *storetab[HASHSIZE];
```

Jun 30 13:14 1987 pns0578 train.h

```
#define STACKSIZE 1000
#define MAXDATA 20
#define false 0
#define true 1
#define IDCMX 10

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

struct {
  int datatype;
  union {
        int ival;
        char *strval;
        } uval;
      } in[MAXDATA];
```

Jun 30 13:14 1983 pns0578 tdec.h

```
#define readin 700
#define writeout 710
#define iddec 720
#define rprm 730
#define asgn 740
#define putin 750
```

Jun 30 13:14 1983 pns2578 tquard.h

```
struct quard{
        int type;
        union {
              int numval;
              char *idval;
              } term;
        } ;

struct aquard{
        int Cp;
        struct quard *Arg1;
        struct quard *Arg2;
        struct quard *Rsl;
        } ;



#define MAXQUARD 10000
#define Int 900
#define Str 910
#define IDMAX 20
#define STACKSIZE 1000
#define WSIZE 1000
#define SPACE (printf(" "));)
```

Jun 30 13:14 1983 pns0578 tsym.h

```
#define Int 900
#define Str 910
#define TMP 20
#define HASHSIZE 787
#define MAXTYPE 1000
#define MAXID 20


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

struct symvar {
        int typeaddr;
        int storeaddr;
              };
struct symvar *vartab[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];
```

Jun 30 13:14 1983 pns0578 dftype.h

```
#define SSRANGE 950
#define RECFIL 960
#define INTSUB 990
#define STRSUB 995
#define RCSUB 998
#define RECSUB 999
#define ITEM 955
#define NULL 0
#define MAXINT 99999999
#define MININT 0
#define on 0
#define off 1
```

Jun 30 13:14 1983 pns0578 thh.h

```
#define IN     400
#define CUT    410
#define GOTO   420
#define STOP   430
#define SIMPLE 600


#define MII 50
#define ID   17
#define MAXID 20
#define MAXIF 100
#define true 1
#define false 0


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 ARG1TP quardsave[iq]->Arg1->type
#define ARG2TP 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 RSITP  quardsave[iq]->Rsl->type
#define STRSINM storetab[rslplace]->ustore.storenum
#define STARG1NM storetab[arg1place]->ustore.storenum
#define STARG2NM storetab[arg2place]->ustore.storenum
#define STRSIID  storetab[rslplace]->ustore.storeid
#define STARG1ID storetab[arg1place]->ustore.storeid
#define STARG2ID storetab[arg2place]->ustore.storeid
```

```
#define STRSITP  storetab[rsiplace]->storetype
#define STARG1TP storetab[arg1place]->storetype
#define STARG2TP storetab[arg2place]->storetype
#define SCRT1 typetab[arg1type]->sort
#define SCRT2 typetab[arg2type]->sort
#define SCRTRSI typetab[rsltype]->sort
#define GRCUP1 getvar(ARG1ID); arg1place = get(); arg1type = savetype;
#define GRCUP2 getvar(ARG2ID); arg2place = get(); arg2type = savetype;
#define GRCUP3 getvar(PSIID); rsiplace = 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 ITEMRSIID storetab[STRSINM]->ustore.storeid
#define ITEMRTP storetab[STRSINM]->storetype
#define ITEM1TP storetab[STARG1NM]->storetype
#define ITEM2TP storetab[STARG2NM]->storetype
#define BADMESS printf(" Comparison 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 *install();
extern struct nlist *idtab[HASHSIZE];
extern struct symtype *typetab[MAXTYPE];
extern struct quard *quardsave[MAXQUARD];
extern maxstk, exter; stackptr;   extern addr;     extern nmerrs;
extern pcptype();  extern pushtype();     int PCPAPE false;
extern poplt();    extern pushlt();     extern popub();   extern pushut();
extern poparr();   extern pusharr();   extern arrptr;
extern pusharrfst();   extern pcparrfst();
extern pusharrsize();  extern pcparrsize();
extern typestk[20];    extern typeptr;    extern maxtype;
extern fixstk[20];     extern fixptr;     extern maxfix;

extern dumptype();

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

char *varsave[MAXID];
char *strtemp, *newtemp();
char *newindtmp();
char *idsave[MAXID], *string[MAXID];
char *strtmpsv, *strtempM;
char *iosave;
char *idtemp;

int tmp;
```

```
int arraystore = 1;
int numplace;
int D[MAXID], DC = 0;
int XX;
int fig;
int itemcount=0;
int PK;
int notfix = true;
int getflag = false, notfirst = false;
int listsave=0;
int sub=0, temp;
int firsttmp, chaintmp, ad;
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[MAXIF];
int ii;
int idcount=0;
int ub[MAXID], lb[MAXID], arraysize=1, wordcount;
int numtype = 1, numvar = 0, sccnt = 0;
int firststore[MAXID], save;
int cin = 0, st;
int index = false, ind;
int testfirst=true;
int popsave[MAXID];
int wcsve[MAXID];
int firstrec[MAXID];
int fr = -1, rc = 0, wc = 0;
int iisave = 0, isave, jj=0;
int morearray=true, firstarray=true, TYPE,   typearray, typetmp, arrfst;
int PCOUNTS=1, array=false;
int changearrsize=false;
int arrsve[20], ac=0, arraycount=0;
int idfirst=true;
```

Jun 30 13:14 1983 pns0578 th.h

```
#define MII 50
#define II   17
#define MAXID 20
#define MAXIF 100
#define true 1
#define false 0


typedef struct {
        char *id[MAXID];
        int num;
        char *slen;
        int tp;
        int vp[MAXID];
        int place;
              } irs;
#define YYSTYPE irs

int vartmp;
```

Jun 30 13:14 1983 pns0578 tsim.h

```
#define true 1
#define false 0


#define ARG1NUM  quardsave[iq]->Arg1->term.numval
#define ARG2NUM quardsave[iq]->Arg2->term.numval
#define ARG1TP quardsave[iq]->Arg1- type
#define ARG2TP 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 RSLTP  quardsave[iq]->Rsl->type
#define STRSLNM storetab[rslplace]->ustore.storenum
#define STARG1NM storetab[arg1place]->ustore.storenum
#define STARG2NM storetab[arg2place]->ustore.storenum
#define STRSLID  storetab[rslplace]->ustore.storeid
#define STARG1ID storetab[arg1place]->ustore.storeid
#define STARG2ID storetab[arg2place]- ustore.storeid
#define STRSLTP  storetab[rslplace]->storetype
#define STARG1TP storetab[arg1place]->storetype
#define STARG2TP storetab[arg2place]- storetype
#define SORT1 typetab[arg1type]->sort
#define SORT2 typetab[arg2type]- sort
#define SORTRSL typetab[rsltype]- sort
#define GROUP1 getvar(ARG1ID); arg1place   get(); arg1type = savetype;
#define GROUP2 getvar(ARG2ID ; arg2place   get(); arg2tyle = savetype;
#define GROUP3 getvar(RSLID); 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 ITEMRSLNM storetab[STRSLNM]->ustore.storenum
#define ITEMRSLID storetab[STRSLNM]->ustore.storeid
#define ITEMRTP storetab[STRSLNM]->storetype
#define ITEM1TP storetab[STARG1NM]- storetype
#define ITEM2TP storetab[STARG2NM]->storetype
#define PARMESS printf(" Comparison 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 *install();
extern struct nlist *idtab[HASHSIZE];
extern struct symtype *typetab[MAXTYPE];
extern struct guard *guardsave[MAXGUARD];
extern maxstk, exter; stackptr;   extern addr;     extern nmerrs;
extern poptype();  extern pushtype();     int PCPARR=false;
extern popit();    extern pushit();     extern poput();   extern pushub();
extern poparr();   extern pusharr();   extern arrptr;
extern pusharrfst();   extern poparrfst();
extern pusharrsize();  extern poparrsize();
extern typestk[20];    extern typeptr;    extern maxtype;
extern fixstk[20];     extern fixptr;     extern maxfix;

extern dumptype();

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

char *varsave[MAXID];
char *strtemp, *newtemp();
char *newirdtmp();
char *idsave[MAXID], *string[MAXID];
char *strtmpsv, *strtempM;
char *iosave;
char *idtemp;

int tmp;
int arraystore = 1;
int numplace;
int I[MAXID], IC = 0;
int XX;
int fig;
int itemcount=0;
int PK;
int notfix = true;
int getflag = false, notfirst = false;
int listsave=0;
int sub=0, temp;
int firsttmp, chairtmp, 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], endiffla[MII], ifcount[MAXIF];
int ii;
int idcount=0;
int ub[MAXID], lb[MAXID], arraysize=1, wordcount;
int numtype = 1, numvar = 0, sccurt = 0;
int firststore[MAXID], save;
int cin = 0, st;
int index = false, ird;
int testfirst=true;
int popsave[MAXID];
int wcsve[MAXID];
int firstrec[MAXID];
int fr = -1, rc = 0, wc = 0;
int iisave = 0, isave, jj=0;
int morearray=true, firstarray=true, TYPE,   typearray, typetmp, arrfst;
int FOUNDS=1, array=false;
int changearrsize=false;
int arrsve[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 :

Name of Mini-language	Directory name
Mini-language Core	Core\
Mini-language D	D\
Mini-language Type	Type\
Mini-language Procedure	proc\

The various directories are accessed by typing

```
cd Directory-name
```

for example

```
cd Core
```

will get one 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 3

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

Rs1 ABC = 5

This means that the result value of variable 'ABC' is an

integer 5, or

Rsl ABC 'true'

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

Rsl ABC = true

which means the result value of variable 'ABC' 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 MAXGUARD in 'guard.h'

appears on the screen, it means that the constant number given, 'MAXGUARD' 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 'guard.h' and change the 'MAXGUARD' value to a bigger one.

(2) Make the user program smaller.

Rochester Institute of Technology
School of Computer Science and Technology

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