835

A PASCAL COMPILER FOR A STACK BASED MACHINE

648

Dissertation Work Submitted to

Jawaharlal Nehru University

in partial fulfilment of the requirements for the award of the Degree of Master of Technology in Computer Science and Technology

by

SUNIL RAINA

SCHOOL OF COMPUTER AND SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI - 110 067
JANUARY 1991

CERTIFICATE

This is to certify that the dissertation entitled **"A PASCAL Compiler For a Stack Based Machine"**, being submitted by me to Jawaharlal Nehru University in the partial fulfilment of the requirements for the award of the degree of **Master of Technology**, is a record of original work done by me under the supervision of **Dr. P. C. Saxena**, Associate Professor, School of Computer and Systems Sciences, Jawaharlal Nehru University during the year 1990, Monsoon Semester.

The results reported in this dissertation have not been submitted in part or full to any other University or Institute for the award of any degree or diploma, etc.

[illegible]

SUNIL RAINA

[illegible]

Prof. N. P. Mukherjee
Dean,
School of Computer and
Systems Sciences,
J.N.U.,
New Delhi.

[illegible]

Dr. P. C. Saxena
Associate Professor,
School of Computer and
Systems Sciences,
J.N.U.,
New Delhi.

ACKNOWLEDGEMENT

I owe my sincere thanks to my supervisor **Dr. P. C. Saxena** for his uncompromising guidance, constant supervision and constructive criticism without which this work would not have been completed successfully.

I extend my sincere thanks to **Prof. N. P. Mukherjee**, Dean, School of Computer and Systems Sciences, Jawaharlal Nehru University for his encouragement and facilities provided for the completion of this work.

I also take this opportunity to thank all faculty and staff members and the friends who helped me in every way possible.

SUNIL RAINA

C O N T E N T S

PREFACE

The apellation 'stack computer' designates a class of computers using one or more stacks. Over the years a number of such machines have appeared, and that includes the Burroughs B5500,B5700,B6700 etc. Even the conventional machines incorporate some sort of elaborate stack mechanisms, along with the usual hardware. It is felt that computers, operating systems and programs in general are both easier to write for and run more comfortably than on conventional computers.

We have worked on the design, analysis and implementation of a Pascal Compiler for such a stack based machine. Due to shortage of time only a working subset of Pascal has been taken for consideration. The language features two pre-defined data types integer and real, standard I/O procedures, expressions through binary arithmetic and relationship operators. It is a block type language following Pascal type scope rules and the procedures may be recursive. All the parameters may be passed by reference unlike Pascal where both call by value and call by reference are allowed. The object code is composed of instructions capable of directly executing on a stack based architecture.

Two areas where the stack based machines are at variance with the conventional machines are STORAGE EFFICIENCY and MEMORY REFERENCE EFFICIENCY. Storage efficiency which refers to the compactness of the encoding of programs with the idea that the instruction sets are efficient, is definitely a gainer as far as stack based machines are concerned. Memory reference efficiency generally refers to the amount of memory actually referenced to execute a particular instruction-set. It is believed that stack based machines have a poor efficiency in this field. We tried to indicate the reasons for these views and found the strengths and weaknesses of stack designs.

CHAPTER 1

COMPILERS

Interactions involving humans are most effectively carried out through the medium of language. Language permits the expression of thoughts and ideas, and without it, communications, as we know it, would be difficult.

A programming language serves as a means of communication between the person with a problem and the computer. A program solution to a given problem will be easier and more natural to obtain if the programming language used is close to the problem. Such a programming language is usually high level.

The hierarchy of programming languages based on increasing machine independence includes the following:

1. Machine level languages
2. Assembly language
3. Higher level or user oriented languages
4. Problem-oriented languages.

1. A machine-level language is the lowest form of computer language. Each instruction in the program is represented by numeric code and numeric addresses are used throughout the program to reference the memory.

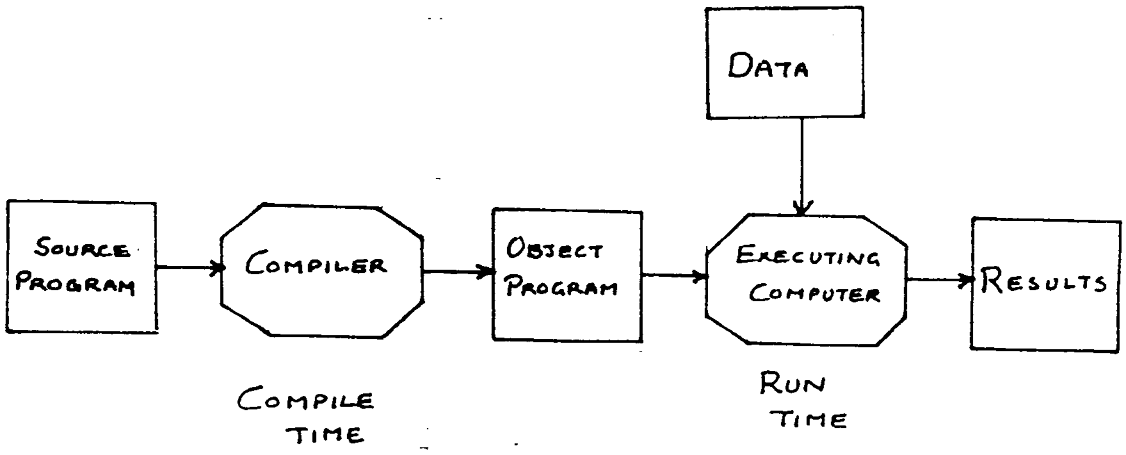
2. Assembly language is essentially the symbolic version of the machine level language.

3. A high-level language offers most of the features of the assembly language in addition to a set of features like structured control constructs, nested statements, blocks, procedures etc.

4. A problem-oriented language provides for the expression of the problem in a specific application or problem area.

TRANSLATORS

A translator inputs and then converts a source program into an object or target program. The source program is written in the source language and the object program belongs to an object language. A translator which transforms a high level language such as FORTRAN, Pascal, COBOL into a particular computer's machine or assembly language is called compiler.



The Compiliation Process

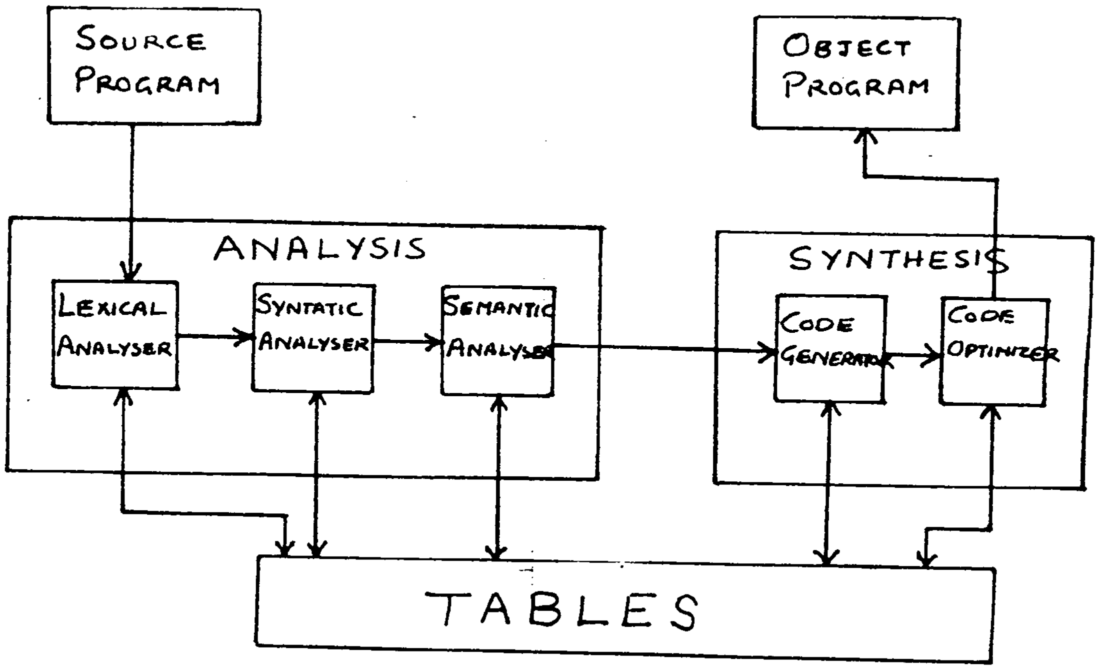
The Tasks of a Compiler

The compilation is usually implemented as a sequence of transformations (SL,L1), (L1,L2) . . . (Lx,TL). Where SL is the source language and TL is the target language. Each language Li is called an intermediate language. Intermediate languages are conceptual tools used in decomposing the task; i.e. Compiling from the source language to the target language.

Any compilation can be broken down into two major tasks:

*<u>Analysis</u>	:	Discover the structure the primitives of the source program, determinging its meaning.
*<u>Synthesis</u>	:	Create a target program equivalent to the source program.

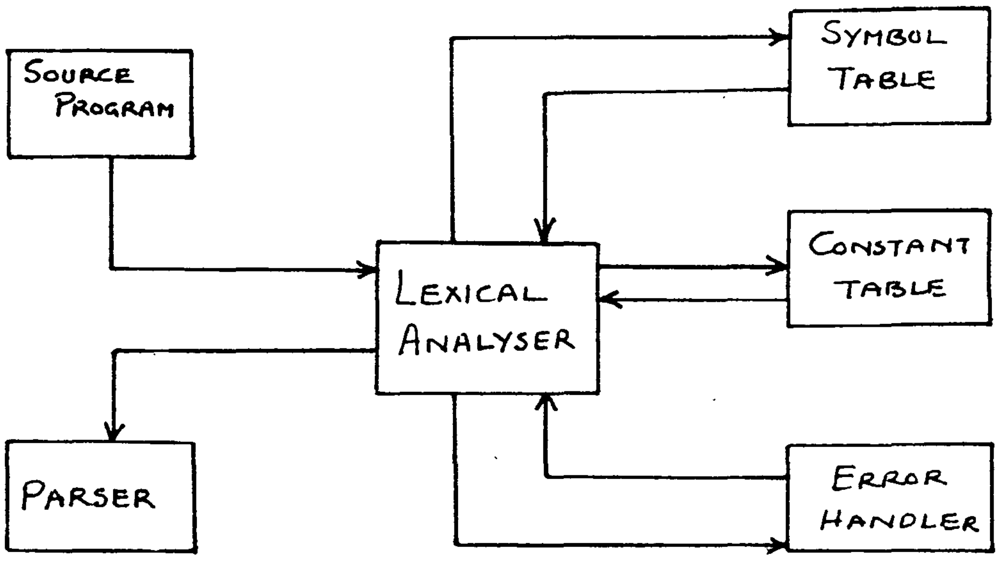
<u>COMPONENTS OF A COMPILER</u>



The analysis concerns itself solely with the properties of the source language. It converts the program text submitted by the programmer into an abstract representation embodying the essential properties of the algorithm. The analysis phase may be divided into three sections - the lexical analyser, parser and the intermediate code generator.

(i) Lexical analyser -

The lexical analyser converts the source program from a character string to a sequence of semantically relevant symbols. The symbols and their encoding form the intermediate language output from the lexical analyser. The lexical analyser interfaces with the other modules of the compiler as shown in the following figure:



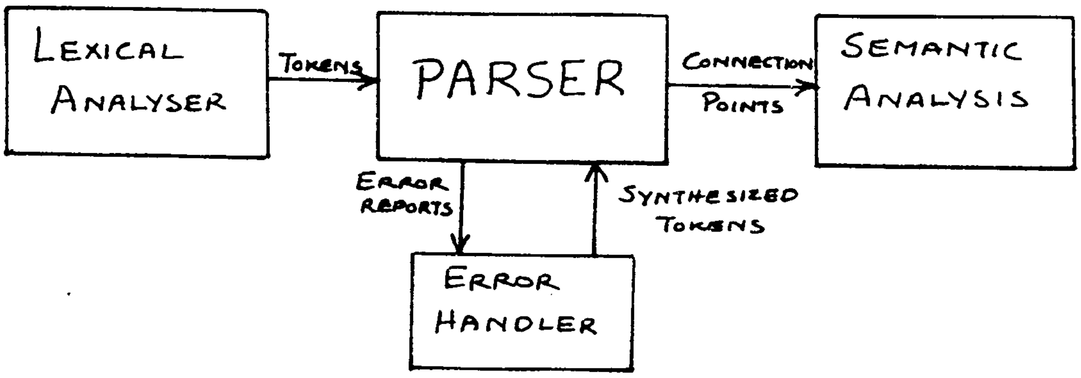
THE LEXICAL ANALYSER

The lexical analyser reads the input text indentified as a basic symbol. After this identification, the lexical analyser either creates a "token" describing it or restarts in a new state. Thus formed tokens are passed on to the parser to form grammatically correct sentences out of these tokens. These tokens are also placed in the symbol or constant tables accordingly.

If lexical errors such as unrecognized input characters and violations of the basic symbol grammar are encountered, the error handler is invoked which determines the continuation of the lexical analysis.

(ii) Parser:

The parsing of a source program determines the semantically-relevant phrases and, at the same time, verifies syntactic correctness. As the output of the parser, we obtain a parse tree of the program. Besides, the parser interfaces with other modules of the compiler as follows:



A PARSER'S ENVIRONMENT

The parser accepts a sequence of basic symbols, recognizes the extant syntactic structure, and outputs that structure along with the identity of relevant symbols. If the syntactic structure of the input text is not error free, the parser invokes the error handler to report errors and to aid in recovery so that processing can continue.

(iii) Intermediate Code Generator:

An intermediate source form is an internal form of a program created by the compiler while translating the program from a high-level language to assembly-level or machine-level code. There are many advantages of using the intermediate source forms. For example, it represents a more attractive form of target code than the assembly or the machine code. Besides, certain optimization strategies can be more easily performed on intermediate source forms than on either the original program or the assembly-level or machine-level code.

The compilers which produce a machine independent intermediate source form are more portable than those which do not.

However, there is a certain disadvantage associated with the intermediate code forms which is that code produced can be less efficient than producing machine-level code directly. The argument behind this is that an intermediate language necessitates another level of translation(i.e. from the intermediate source form to the machine level code).

The intermediate code generator takes the parse tree provided by the parser as the input and generates equivalent intermediate source forms such as polish notation, n-tuple notation, abstract syntax trees, threaded code and pseudo or abstract machine code.

Synthesis proceeds from the abstraction developed during the analysis phase. It augments the intermediate code by attaching additional information that reflects the source-to-target mapping.

The synthesis consists of two distinct subtasks - code generation and assembly. The code generation transforms the abstract source program appearing at the analysed synthesis interface into an equivalent target machine program. This transformation is carried out in two steps: first we map the algorithm from source concepts to target concepts, and then we select a specific sequence of target machine instructions to implement the algorithm.

Assembly resolves all target addressing and converts the target machine instructions into an appropriate output format. By the term "assembly" we do not imply that the code generator will produce symbolic assembly code for input to the assembly task. Instead, it delivers an internal representation of target instructions in which most addresses remain unresolved. The output of the assembly task should be in the format accepted by the standard link editor or loader on the target machine.

(i) Code Generator :

The code generator creates a target tree from a structure tree. This task has three sub-tasks:

a) Resource allocation: Determine the sources that will be required and used during execution of instruction sequences.

b) Execution order determination: Specify the sequence in which the descendants of a node will be evaluated.

c) Code Selection: Select the final instruction sequence corresponding to the operations appearing in the structure tree.

In order to produce code optimum under a cost criterion that minimises either program length or execution time, these subtasks must be intertwined and iterated.

(ii) Assembly :

The task of the assembly is to convert the target tree produced by the code generator into the target code required by the compiler specification. This target code may be a sequence of bit patterns to be interpreted by the control unit of the target machine, or it may be text subject to further processing by a link editor or loader.

Assembly is essentially independent of the source language and should be implemented by a common module that can be used in any compiler for the given machine. To a large extent, this module can be made machine independent in design.

During the entire compilation process of analysis and synthesis, errors may appear at any time. In order to detect as many errors as possible in a single run, repairs must be made such that the program is consistent, even though it may not reflect the programmer's intent. Violations of the rules of the source language must be detected and reported during analysis. If the source algorithm uses concept of the source language for which no target equivalent has been defined in a particular implementation, or if the target algorithm exceeds limitations of a specific target language interpreter, this should be reported during synthesis. Besides, errors must also be reported if any storage limits of the compiler itself are violated.

In addition to the actual error handling, it is useful for the compiler to provide extra information for run-time error detection and debugging. In fact, the task of the error handler is to detect each error, report it to the user, and possibly make some repair to allow processing to continue. It cannot generallly determine the cause of the error, but can only diagnoze the visible symptoms. Similarly, any repair cannot be considered a correction; it merely neutralizes the symptom so that processing may continue.

The above discussion highlights the major functions a compiler has to perform.

CHAPTER 2

ABSTRACT MACHINES

The emphasis in computer design is on producing compilers that are both portable and adaptable. One of the approaches used is to produce a form of intermediate source code for an abstract machine. The instruction set for this machine should closely model the constructs of the source languages that are to be compiled.

Portability and abstract machine

A program is said to be portable if it can be moved to another machine with relative ease while a program is adaptable if it can be readily customized to meet several user and system requirements. Suppose a given compiler is to be ported from machine X to machine Y. To realize this, the code generation routines must be rewritten for the machine Y. The task would be much easier if the compiler had been divided into two parts - the front-end dealing with the source code and the back-end dealing with the target machine with a well defined interface. For a well defined interface only the target machine part need be changed.

The flow of information between two parts of a compiler takes the form of language constructs in one direction (front-end to back-end), and the target machine information in the other direction (back to front-end). The interface can be realized by using an abstract machine. The source language constructs can be mapped into pseudo operations on this abstract machine. An abstract machine can be designed for a particular source language (e.g. Pascal in this case).

An abstract machine is based upon operations and modes that are primitive in programming language. The language dependent translator translates the program into abstract machine code by breaking constructs of the language into a sequence of primitive operations on the primitive modes. A primitive mode and a primitive operation form a pair which describe an instruction.

The architecture of the abstract machine forms an environment in which the modes and operations interact to model the language. Unlike a real machine whose architecture is governed by economic considerations and technical limitations, the abstract machine has a structure which facilitates the operations required by the given programming language.

The abstract machine can be embedded into the language dependent translator by a series of interface procedures, one for each abstract machine instruction.

The use of an abstract machine allows the language dependent translation and the machine dependent translator to be seperated by well defined interface. One can refine either part of the compiler without affecting the other.

Another advantage would be the choice it allows in implementing the machine dependent tranlator. The language dependent translator's interface procedures could produce a symbolic assembly code for the abstract machine. The MOT would then be an abstract machine assembler which could be implemented by using a macro processor, either the one provided with the real machine's assembler or a machine-independent macro-processor.

A Standard Abstract Machine

A standard abstract machine is a machine which has been carefully defined around a model that can be used for many programming languages. Many lot's can produce assembly code for the standard abstract machine, and one assembler could be used to translate this assembly code for the target machine.

The standard abstract machine language should be enhancible to allow new operations and modes that appear in a new programming language. The designer of an abstract machine can concentrate on a structure which facilitates the operations in the given source language. The designer must also, however, take into consideration the efficient implementation of the abstract machine on an actual computer.

Advantages of the Abstract Machine

The most important advantage of using an abstract machine is the clean separation of the front-end and the back-end processes of the compiler.

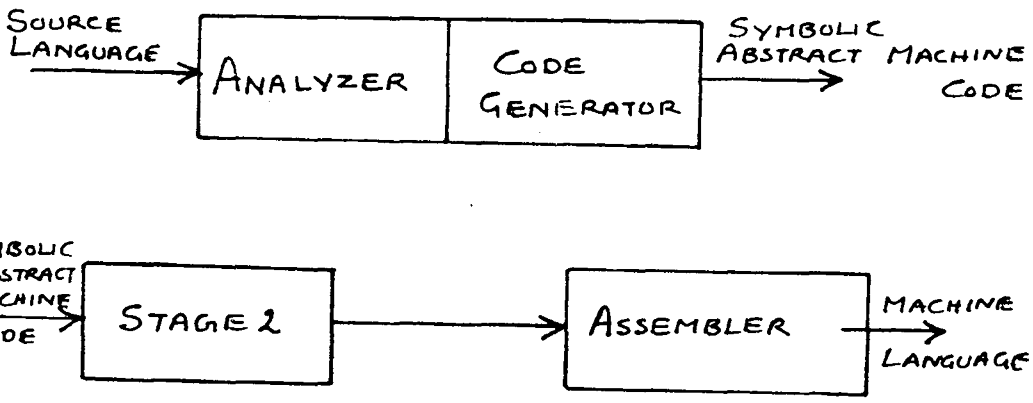
Suppose it is required to implement 'm' distinct languages on 'n' different machines. Without using some form of intermediate source code, such as an abstract machine, m * n different compilers would have to be written, that is one for each language/machine combination. In an abstract machine apporach, however, only m front ends and n back-ends are required. A compiler for a certain programming language and target machine can then be generated by selecting the appropriate front-end and back-end. Using this approach m * n different compilers can be generated from m + n components.

Some Existing Abstract Machines

Today it is quite feasible for a given programming language such as Pascal to produce efficient object code for several different target machines. A more difficult problem is to have an abstract machine from which efficient object code can be produced for several programming languages. The difficulty is to have an abstract machine that efficiently models all these programming languages.

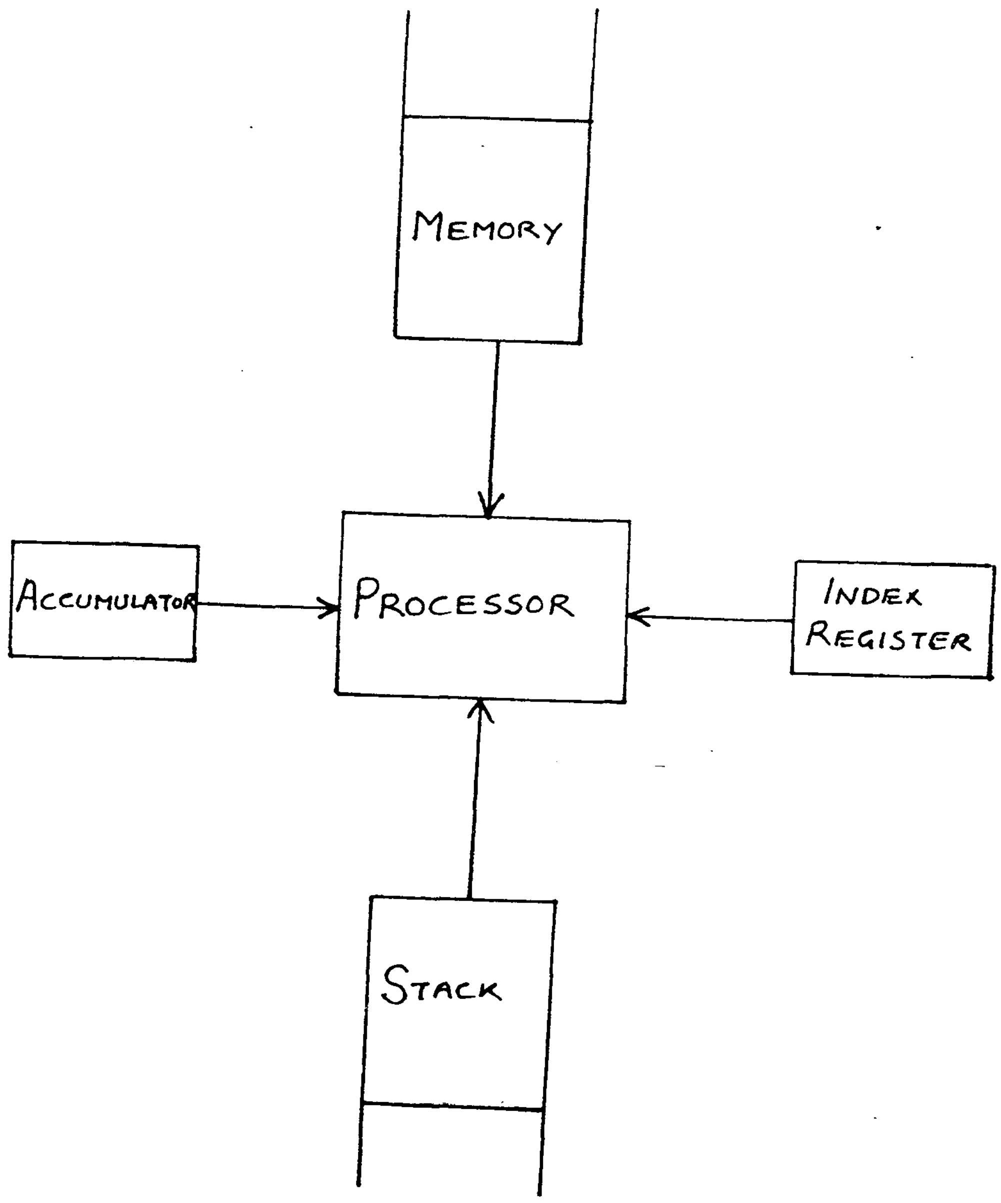
JANUS

A family of abstract machines called 'Janus' has been developed at the University of Colorado in order to study the problems of producing portable software and in particular portable compilers.



The Translation Process

The symbolic Janus code is translated to the assembly language of the target computer by a program such a STAGE. Simple translation rules are supplied by the user to describe the various Janus constructs and the primitive modes and operators. Final translation to object code is provided by the normal assembler of the target computer.



Architecture of the JANUS family of Abstract Machines

It favours computers with a single arithmetic registers or with multiple arithmetic registers and register/storage arithmetic.

2. The IBM S/360 FORTRAN(G) COMPILER

The IBM FORTRAN IV G-level compiler exemplifies an approach to compiler portability in which the abstract machine is implemented on the real machine via an interpreter.

The compiler is written in the language of an abstract machine called POP whose design is well suited to the implementation of compilers. POP is a machine organized around a number of last-in first-out queue i.e. push down stacks. The instructions of the machine are set up to operate on these stacks as well as on a linearly organized memory. There are two stacks, WORK and EXIT which are an integral part of the machine so that the access is efficient.

There are about 100 instructions in the abstract machine, the operation code can be represented in one byte. The operand which represents either a value or a relative address can also be represented in one byte. Since access to WORK and EXIT must be efficient, the pointers to these stacks are maintained in general registers.

The abstract machine called a P-machine is a simple machine-independent stack computer. The language can be easily transported onto a variety of hardware platforms.

The hypothetical stack machine has five registers and a memory. The registers are

PC - Program Counter

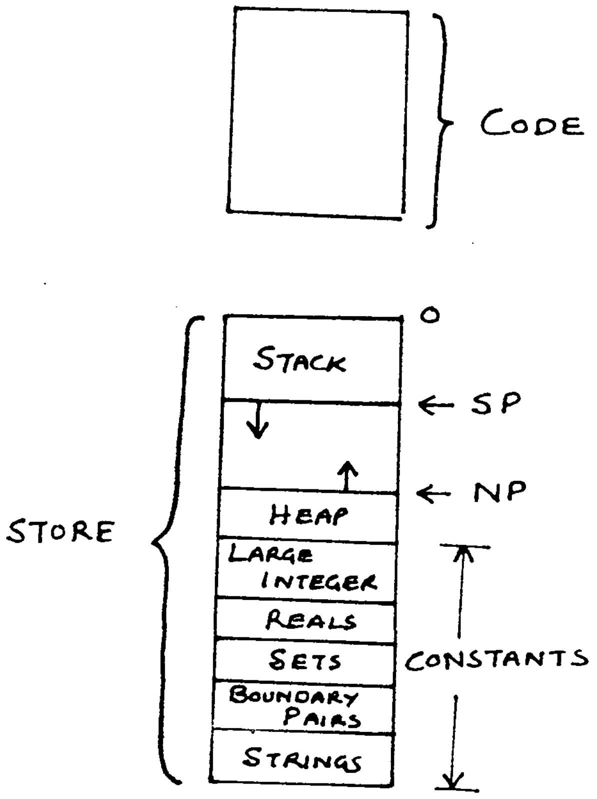
NP - New Pointer

SP - Stack Pointer

MP - Mark Pointer

EP - Extreme Stack Pointer

The last four pointers are associated with the management of storage in memory.



THE ABSTRACT STACK MACHINE.

The memory which can be viewed as a linear array of words is divided into two main parts. The first part of the memory CODE contains the machine instructions. The second part of the memory, the STORE , contains the data (i.e. non instructions) store part of the program. The layout of the computers' memory is as in the figure. PC - refers to the location of the instruction in CODE. STORE contains two parts; the first part represents the various constants of a given program while the second part is dedicated to the other data requirements of the executing program.

The stack whose top element is denoted by SP, contains all directly addressable data according to the data declaration in the program. The heap with associated top element NP, consists of the data that have been created as a result of direct programmer control. The heap is similar to a second stack structure.

The stack consists of a sequence of data segments. Each data segment is associated with the activation of a procedure or a function.

The data segment contains the following sequence of items.

1) A mark stack part.

2) A parameter section.

3) A local data section for any local variables.

4) Temporary elements that are required by the processing statements.

The mark stack part has the following fields:

1. value field for a function
2. static link

The basic modes of PASCAL (e.g. INTEGER and REAL) are supported on the stack compiler. There are several classes of instructions - arithmetic, logical and relational.

P-code has been extended to U-code(Perkins and Sites 1979) to facilitate certain kinds of code optimizations.

CHAPTER 3

STACK COMPUTERS

A stack is a last-in first-out data structure and the simplest operations that can be performed on it are PUSH and POP. A PUSH operation pushes new data onto the top of the stack while a POP operation removes the most recently pushed data item.

The appellation 'stack computer' designates a class of computers using one or more stacks. In the actual implementation, a number of conceptually different stacks are mixed into one tightly-bound interleaved structure.

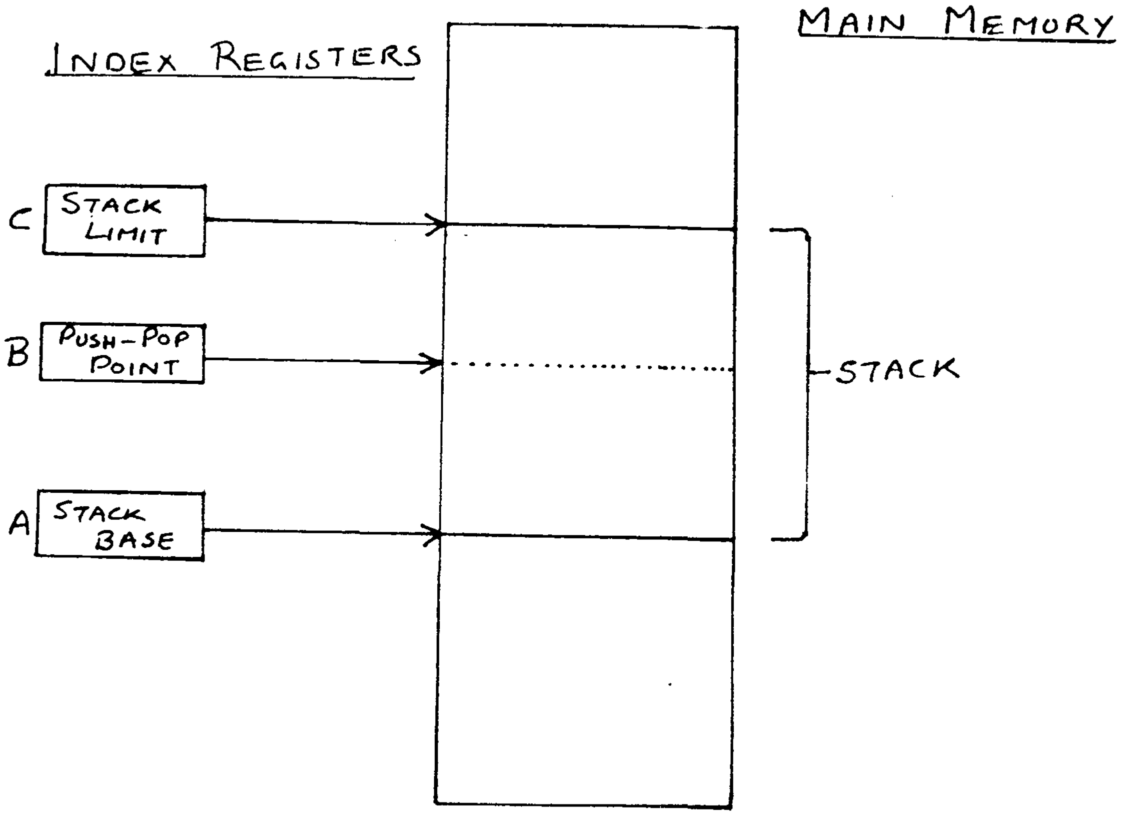
Much work has been done on the concept of stack computers which includes papers by Bauer, Randell, Russel, et. al. One of the earlist stack based machines to come into existance was KDF9 computer. Commercial stack based computers followed with B5000 Burroughs being among the first ones to be followed by B5500,B5700,B6700. Hewlett-Packard HP3000 also supports a stack mechanism and so does Burroughs B1700 with a stack mechanism controlled by a writable microcode. Further DEC'S PDP11 series also supports some basic features of a stack machine.

3616

In a stack machine, a datum of width n is visualized as a contiguous vector of n independent bits of information, thus it can take only 2^n values. A datum is the basic unit of transactions between the various parts of the computer. The data pushed onto the stack comes from somewhere in the computer; the data popped goes somewhere. Each action involving PUSH and POP takes the form of an assignment. An assignment to a stack implies a PUSH onto the stack, an assignment from a stack implies a POP operation.

STACKS

Simple stacks can be implemented in a variety of ways; the more complex versions are generally restricted to being placed in main memory and being accessed through index registers.



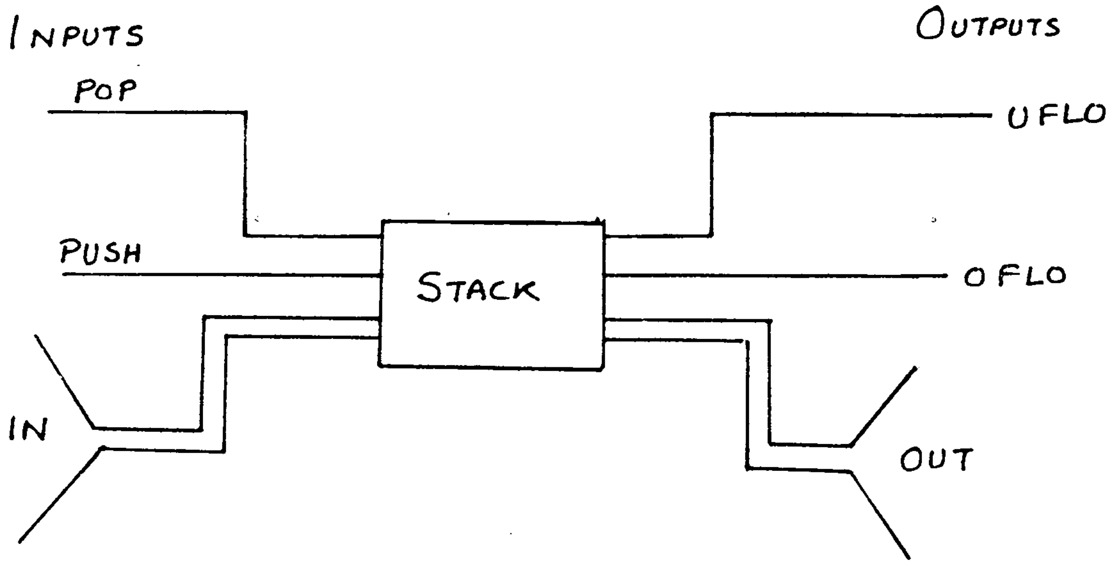
The actions PUSH,POP can be defined in terms of A,B,C and the memory.

Action	Definition

PUSH X	IF B>=C THEN OVERFLOW
	ELSE MEMORY[B]:=X; B:=B+1;

POP X	B := B - 1;
	IF B<A THEN UNDERFLOW
	ELSE X:=MEMORY[B]

A Simple Stack Mechanism



In this scheme, the PUSH and POP are control singnals that cause the corresponding actions to take place. The datum must be present at the IN prior to PUSHing. Output signals UFLO and OFLO correspond to stack failure indications.

The PUSH and POP variants used for the hypothetical stack machine visualized in thit project are as follows:

PushCI, PushI, PopI,PushgI, PopgI, fetchI, popI,pusha, pushga.

All these operations either exchange data between the I/O device and the stack or between the system part of the stack pointed to by the frame pointer(FP) and the data part of the stack pointed to by SP.

<u>Arithmetic Evaluation Stacks</u>

The concept of arithmetic stacks can be understood by the evaluation of an expression say (2 + 3 * 8). The following primitive sequence of operations can be allowed to take place.

```
push 2

push 3

push 8

mul     --> (popping two topmost operands and pushing
             the results back again)

add     --> (popping two topmost operands and pushing
             the sum back again)
```

The action falls into two classes: placing the operands at the top of the stack and operating on the operands at the top. The operations can be accomplished by the use of reverse polish, or postfix which would look like 238*+ in our case.

Conventional computers accomplish arithmetic evaluation with registers instead of stacks. The use of stacks eliminated the need for explicit temporary stores and associated book-keeping.

The size of an arithmetic stack determines how complex an expression can be computed. A difficulty arises when different types of data are mixed (say integer and real). Here the data may have different widths and bit patterns and different interpretations.

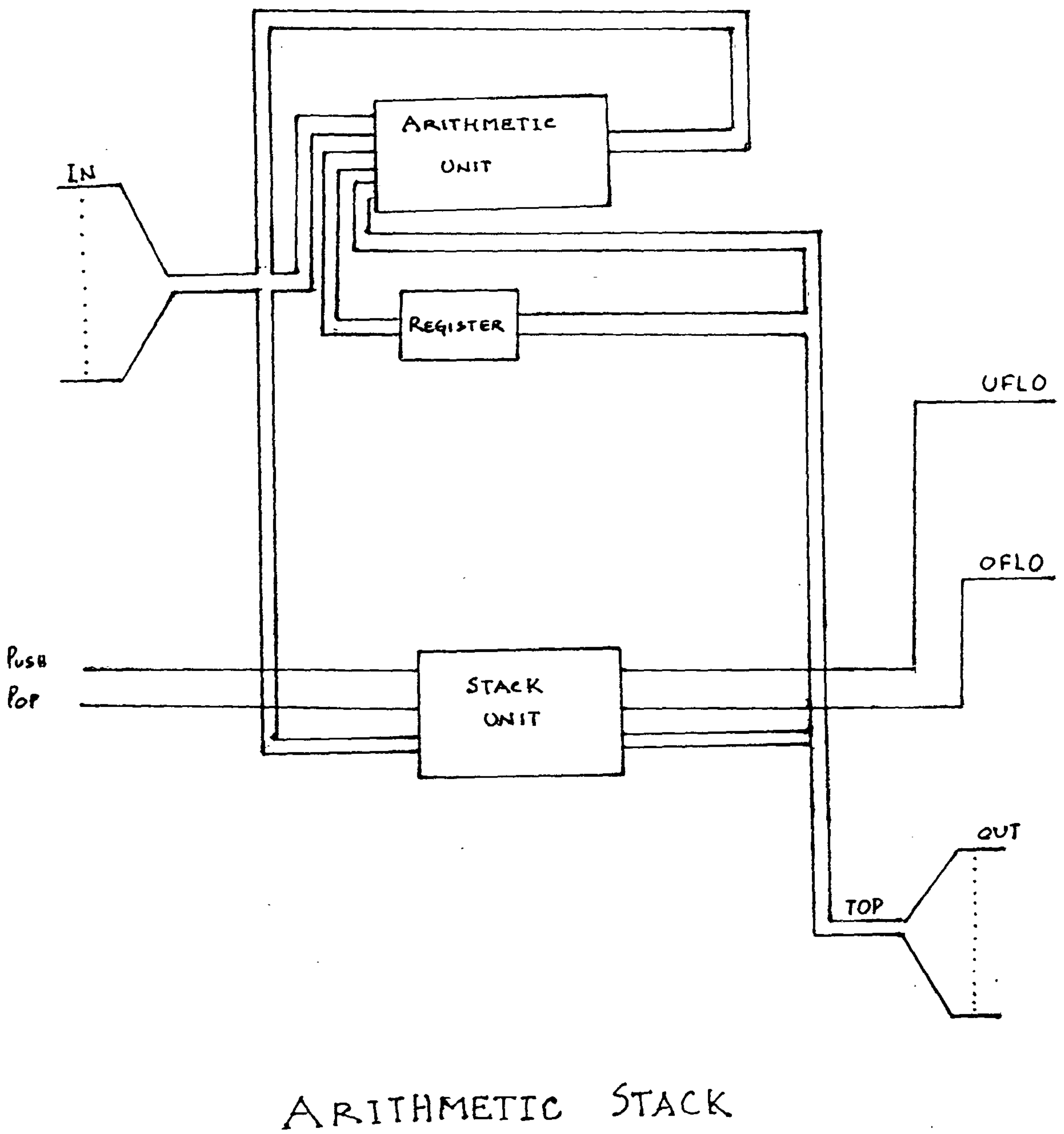
One solution to this problem is to tag the operations(i.e. have different operators for different types of data e.g. "$+_r$" for real addition and "+" for usual addition).

A second approach is to provide a separate evaluation stack for different types of data.

Yet another approach would be to use tagged data such that only when the operands happen to be of the same type, an operation is carried out otherwise a conversion operation converts the dissimilar one to the usual format.

Control Stack

During the execution of a program, the machine code resides in the main memory and is pointed to by a register called program counter (PC). There are two important control points for subroutine entry and exit. PC must be first set to the entry point and the execution allowed to proceed. When the execution is finished , the PC must be



ARITHMETIC STACK

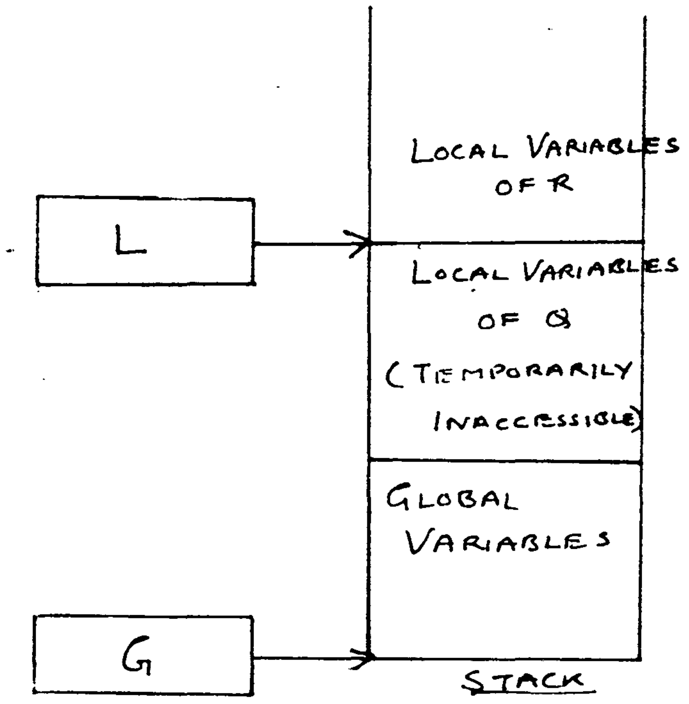
reset to the value it had prior to entry, allowing the calling routine to proceed.

A separate stack is used for accomplishing this, and may be implemented in another part of the memory. The value of PC is saved in the control part of stack and when the souroutine is executed, it may be popped off from the top of the control stack. A stack can be used for the precise reason that CALL and RETURN pairs are nested in time. A distinct advantage in in this mechanism is that subroutine may call each other to any depth and in any order. With no more storage used than is actually needed.

Storage for Simple Variables

The local variables of a subroutine have a property that they can be accessed only from statements within the subroutine.

When the definition of one subroutine is nested within the definition of another, the inner subroutine has access to the variables local to the containing subroutine but not vice-versa. Storage need not be allocated to the local variables until the subroutine is called and may be freed for other uses as soon as control has left the subroutine. This is accomplished by using a stack for local variables. The local variables are not accessed by PUSH and POP , but at random at any time during the execution of their scope.



Global variables are at the bottom of the stack pointed to by G and are accessible to all. Register L points to the base of the area which contains most local variables of the subroutine being currently executed.

CHAPTER 4

Description of the Hypothetical Stack Machine

The machine is composed of the following:

1. There is a word-addressable memory partition for data called 'stack'.
2. Word-addressable memory partition for the program called 'Code'.
3. There are three special purpose registers:

   PC - Program Counter register

   SP - Stack Pointer register

   FP - Frame Pointer register
4. Arithmetic, Logic Unit (ALU), capable of performing various Stack Operations.
5. I/O unit, capable of reading/printing.

The memory partition code contains a program (the byte code of the program written in the corresponding Assembly Language). The memory partition stack is used for storing data and other necessary information.

The Assembly Language

Each line of the Assembly language for the hypothetical stack machine must contain either a comment, a label definition, or an instruction (micro-instruction).

Comments - A Comment begins with a number sign(#) in the first column and extends until the end of the line. These comments are ignored.

Labels : A label is an identifier, and begins with a ($) sign. There must be exactly one definition of every label. The label 'main' indicates the main entry point.

Instructions: An instruction is composed of an instruction name(mnemonic), optionally followed by zero, one or two operands. The mnemonic and operands (if they are present) are separated by space(s) and/or tab(s). An instruction must begin in a column higher than one; otherwise they are treated as a label definitions.

The Instruction Set

CI	- represents an integer constant,
CR	- represents a floating point constant,
L	- represents a label; &L-label; address,
O	- represents a memory offset,
N	- represents an integer constant, which represents either stack frame size, number of parameters, or a stack level (nesting), difference and
C	[illegible]

Opcode	Name	Arg	Action
0	pushcI	CI	SP:=SP+1;Stack[SP]:=CI;
1	pushI	O	SP:=SP+1;Stack[sp]:=Stack[FP+o];
2	popI	O	Stack[FP+O]:=Stack[SP];SP:=SP-1;
3	pushgI	N,O	SP:=SP+1;Stack[SP]:=Stack[base(N)+O];
4	popgI	N,O	Stack[base(N)+O]:=Stack[SP];SP:=SP-1;
5	fetchI		Stack[SP]:=Stack[Stack[SP]];
6	popiI		Stack[Stack[SP-1]]:=Stack[SP];SP:=SP-2;
7	pusha	O	SP:=SP+1;Stack[SP]:=FP+O;
8	pushga	N,O	SP:=SP+1;Stack[SP]:=base(N)+O;
10	addI		Stack[SP-1]:=Stack[SP-1]+Stack[SP];SP:=SP-1;
11	subI		Stack[SP-1]:=Stack[SP-1]-Stack[SP];SP:=SP-1;
12	mulI		Stack[SP-1]:=Stack[SP-1]*Stack[SP];SP:=SP-1;
13	divI		Stack[SP-1]:=Stack[SP-1] div Stack[SP];SP:=SP-1;
14	negI		Stack[SP]:= -Stack[SP];
40	eqI		Stack[SP-1]:=ord(Stack[SP-1]=Stack[SP]);SP:=SP-1;
41	neI		Stack[SP-1]:=ord(Stack[SP-1]<>Stack[SP]);SP:=SP-1;
42	ltI		Stack[SP-1]:=ord(Stack[SP-1]<Stack[SP]);SP:=SP-1;
43	leI		Stack[SP-1]:=ord(Stack[SP-1]<=Stack[SP]);SP:=SP-1;
44	gtI		Stack[SP-1]:=ord(Stack[SP-1]>Stack[SP]);SP:=SP-1;
45	geI		Stack[SP-1]:=ord(Stack[SP-1]>=Stack[SP]);SP:=SP-1;
60	jumpz	L	if Stack[SP]=0 then PC:=&L;SP:=SP-1;
61	jumpnz	L	if Stack[SP]<>0 then PC:=&L;SP:=SP-1;
62	jump	L	PC:=&L;
70	enter	N	Stack[SP+1]:=base(N);Stack[SP+2]:=FP;SP:=SP+3;
71	alloc	N	SP:=SP+N;
72	call	L,N	FP:=SP-(N+2);Stack[FP+2]:=PC;PC:=&L;
73	return		SP:=FP-1;PC:=Stack[FP+2];FP:=Stack[FP+1];

Floating Point Instructions

Floating point instructions are obtained by replacing the last letter 'I' with 'R'. For example, 'pushR' corresponds to 'pushI'. Opcodes of the floating point instructions are always greater by 128 than the opcodes of the corresponding integer instructions. For example, the opcode of 'pushI' is 129. The only additional instructions are 'int'(convert to integer representation), 'intb'(convert to integer below SP), 'fit' (convert to floating point representation; function float changes the type of its argument), and 'ftlb'(convert to float below SP).

Opcode	Name	Arg	Action
15	int		Stack[SP]:=trunc(Stack[SP]);
16	intb		Stack[SP-1]:=trunc(Stack[SP-1]);
144	flt		Stack[SP]:=float(Stack[SP]);
145	fltb		Stack[SP-1]:=float(Stack[SP-1]);

Description of the language for which the compiler is built. (A subset of Pascal with slight variations to make the programming somewhat less tedious and easier to understand). The language is case insensitive except for strong constants.

Comments

A comment is a sequence of characters enclosed within a pair of matching braces ('{' and '}'). Comments can extend over several lines and are ignored as are newline(\n), tab(\t) and white spaces( ).

Tokens

Sequence of characters enclosed within the double quotes (") are literals (ground tokens). Any other sequence of characters denotes a name of a lexical class.

```
letter   ::=   "a" | "b"| . . .|"z"|"A"|"B"|. . .|"Z"
digit    ::=   "0"|"1"|. . . . |"9"
```

Identifiers

An identifier is a finite sequence of letters and digits which begin with a letter. Upper and lower case letters are allowed but there is no distinction between the corresponding lower and upper case letters. Identifiers may be of any length, but only the first ten characters are significant.

identifier ::= letter (letter|digit)*

Numbers

An unsigned integer is a sequence of one or more digits.

unsigned-integer ::= digit digit*

A floating point number is defined as

floating-point ::= unsigned-integer "." unsigned-integer.

String Constants

A string constant is a sequence of characters enclosed within two single quotes. A single quote preceded by a backslash (\) is treated as a character in the string in which it occurs. Similarly, the letter n preceded by a backslash (\n) denotes the newline(ASCII 10) character and two consecutive backslashes (\\) denote a single backslash. A string constant may not extend beyond the end of the line. A { } pair included within a string constant is not treated as a comment. A string constant is used only as an argument of the standard procedure 'writeset'. Also, corresponding upper and lower case letters are considered different with string constants.

```
string-char  ::= ASCII-char|"\\"|"\n"|"\"
string-constant ::= "'" string-char*"'"
```

Operators

```
add-op    ::=  "+"|"-"
mul-op    ::=  "*"|"/"|"div"|
rel-op    ::=  "<"|"<="|"="|"<>"|">"|">="
```

The Grammar

```
program   ------>  "program" identifier ";" block "."
block  ------> declarations "begin" statement_list "end"
declarations -----> declarations declaration|e
declaration  ----->  variable _declarations|
                     procedure_declarations
variable_declaration ---> "var" variable_declaration|
                              variable_declarations
                              variable_declaration.

variable_declaration--->identifier_list":"type_name";"

type_name   -----> "integer"|"real"

procedure_declarations ----> procedure_declaration|
                             procedure_declarations
                             procedure_declaration
procedure_declaration ---> procedure_header block";"

procedure_header   ---->  "procedure" identifier "("
                           parameter_list")"";"|
                           "procedure" identifier";"

parameter_list ---->  parameter_group | parameter_list
                      ";" parameter_group

parameter_group ----> identifier_list ":"type_name

statement_list ----> statement|statement_list ";"
                     statement

statement   ---->  e/* empty statement*/   |
                   identifier|
                   identifier "(" expression_list")"|
                   variable ":=" expression|
                   "while" expression "do" statement_
                    list"end"|
                   "if" expression"then"statement_list
                   "else"statement_list "end"

expression_list---> expression|expression_list":"
                    expression

expression ------> simple_expression |
                   simple_expression rel_op simple_
                   expression

simple_expression----> term|add_op term|
                       simple_expression add_op term

term ----> factor | term mul_op factor

factor ----> variable | constant | "("expression")"

variable----> identifier

constant -----> string_constant | number
```

"," identifier

Data types

There are two standard (predefined) types : integer and real.

Blocks

There are seven predefined identifiers. Integer and real represent two standard data types, readi, readr, writei, writer and writetxt are the names of standard I/O procedures. Each of the I/O procedures takes only one actual parameter. readi, readr can be used to read a single number of types integer or real, respectively. writei, writer may be used to writeout a value of type integer or real respectively. The last procedure writetxt is used to write out a string constant. The language follows usual scope rules of Pascal and the procedures may be recursive.

Expression

There are five binary operators +,-,*,/,div.

div - represents division of integers with integer result.

+ and - have lower precedence than *,/,div

+ and - may also be used as unary operators

Given integer arguments, each operation returns integer results, except for /(division), which always returns a real result. If any of the arguments is of the real type, the result is always of the type real. There are six binary relational operations <,<=,=,<>,>,>=.

Statements

Five types of statements have been defined

- empty statement.
- procedure call
- assignment
- while statement
- if statement

Parameter Passing

All parameters are passed by reference.

PASCAL

Pascal is fairly complete in terms of the programming tools it provides to a programmer, providing a rich set of data types and structuring methods that allow the programmer to define his own data types.

Pascal is a procedure-oriented language. A procedure is the basic unit of a program and each is always defined within another procedure. The main program is considered to be a procedure without parameters which is defined in and called from the software system.

The Abstract Machine for Pascal

The Pascal followed in this project is a subset of the standard PASCAL with minor syntactic modifications.

Standard Identifiers

There are seven predefined identifiers:

integer, real -- predefined data types

writei, writer -- standard output procedure

readi, readr -- standard input procedure

STANDARD PROCEDURES

writei,writer,readi,readr

RESERVED WORDS

PROGRAM, BEGIN END, VAR, INTEGER, REAL, PROCEDURE, WHILE, DO, IF, THEN, ELSE.

OPERATORS

Procedure	Operators
1(highest)	*,/,div
2	+,-
3(lowest)	=,<>,<,<=,>,>=

Scope of Variables

The scope rules followed are the usual Pascal Scope rules. The constants and variables that appear within the action statements of a procedure may have been declared externally, within a program block that contains the procedure itself. Those constants and variables which are declared within a block containing the procedure declaration can be utilized anywhere within the block, whether inside of or external to the procedure. Identifiers defined in this manner are considered to be global to the procedure.

The scope of an identifier refers to the region within which the identifier is declared and can hence be utilized. This applies to all declarations. To transfer information across procedure boundaries is to make use of global identifiers, since global identifiers can be referenced whenever necessary. Procedures can be nested within other procedures but a procedure can not be accessed outside the block that contains the procedure declaration.

CHAPTER 5

PROGRAM DESCRIPTION

Module Lex.C

This module contains the main program called main(argc,argv) which is invoked when the program starts executing. argc is an integer defining the number of arguments on the command line, argv is an array of characters holding all words on the command line. This would generally be the name of the program to be compiled. The input filename must be of the form filename.u and the maximum length of the filename can be of only 15 characters with the rest of the characters being ignored by the compiler. An output file of the name filename.asm is opened in the write mode and object code is put into this file. Control is transferred to the module PAR-C after obtaining a token by the routine getsym(). Finally both the files are closed and a success message (--Happy Halt--) is displayed. An error message is displayed in case of an error.

Module Main.C

This module contains functions which generate tokens for use by the parser module. The various functions within this module are:

1. getsym() - this returns a token and in case of an error

passes the control to the error module through print-tok(tok) where tok is of type token.

2. getsym2() - This function again returns a token and is invoked by the function getsym(). It strips out all blanks'( )', tabs '\t' and newlines '\n' from the input program file, at the same time keeping track of the line no (by counting the number of newline characters '\n') to be used by the error routines in case of an error. All the comments within '{}' pair of braces are ignored. If an EOF char is encountered in the comments, an error message is displayed.

a) All numeric constants without any intervening spaces are separated out as integer and real constants(reals or floating point constants are defined as integer constant followed by decimal point, followed by an integer constant). The two tokens returned in this case are

---- float_sym

---- int_sym

b) In case of relational operators, it looks for '=','<','>' or a combination of these and the various tokens returned are

---- lt_sym (less than symbol)

---- eq_sym (equal to symbol)

---- gt_sym (greater than symbol)

---- lteq_sym (less than or equal to symbol)

---- gteq_sym (greater than or equal to symbol)

---- noteq_sym (not equal to symbol)

c) In the case of a string of alphanumeric characters starting with an alphabet, a token type of identifier is returned. In case the identifier matches with one of the reserved words of the language as defined in the declaration keyword[], tokens of following types are returned.

begin	---	returns begin_sym
dir	---	returns dir_sym
do	---	returns do_sym
else	---	returns else_sym
end	---	returns end_sym
if	---	returns if_sym
integer	---	returns integer_sym
procedure	---	returns procedure_sym
program	---	returns program_sym
real	----	returns real_sym
then	----	returns then_sym
var	----	returns var_sym
while	----	returns while_sym

identifier ---- returns ident_sym (could not find a matching key word)

d) If the tokens are relational operators or parenthesis, following tokens are returned.

(	-	returns lparen_sym
)	-	returns rparen_sym
*	-	returns mult_sym
+	-	returns a plus_sym
,	-	returns comma_sym
-	-	returns minus_sym
..	-	returns period_sym
/	-	returns divide_sym
:=	-	returns assign_sym
;	-	returns semicolon_sym

3. <u>Lower(c)</u>

This function takes any uppercase alphabetic character and returns the corresponding lowercase character. In this manner, this version of Pascal makes no difference between uper and lower case alphabets.

<u>Module par.C</u>

This is the parser module and contains a number of functions which parse the input program in accordance with the defined grammar. The different functions used in this module are as follows.

1. Program (sset)

Control is handed over to this function by the lexical module after getting a token from the token generator. The input sset is of type set which is defined of the unsigned integer. The function installs the symbol table and starts off with the first token which should be program_sym and should be followed by ident_sym (i.e. an identifier) and a semicolon_sym. Next the function block is invoked and in case of an error each stage will send out an unique error-message.

2. Block (sset, no-var,no-para,proc-ptr)

This function looks for the declarations by invoking the function declarations and looks for the begin_sym. Function statement_list is invoked thereafter and if end_sym is found after that control is returned back to program.

3. decls(sset, no-var,no-para)

Until a begin_sym or a procedure_sym is found, this function stores all the declarations following the var_sym adding them to sset.

4. var-decls (sset,no-var), var_decl(sset, no-var)

These two functions are invoked by function decls look for the identifier_list.

5. Proc-decl(sset,no-para)

This function looks for procedure defined with the main program followed by its own separate block.

6.proc-header(sset,no-para,proc-ptr)

Procedure_sym is looked for when the function takes over followed by the procedure name in the form of an identifier which may or may not be followed by a parameter list enclosed within left and right parenthesis.

7.Parameter-list(sset, no-para)

This function looks for a parameter_group until a semicolon is obtained.

8.Parameter-group(sset,no-para)

This function is invoked by the parameter_list function and looks for an identifier_list followed by a colon_sym and a type_name.

9.Statement list(sset)

This function looks for a statement followed by a statement_list with a semicolon_sym as a delimiter. Control is passed on to function statement.

10.Statement(sset)

A statement in this language may consist of either assignment statements or control statement. If the first symbol is ident_sym, then the symbol table is searched to see if the identifier has been declared at the beginning . An optional expression_list is looked for within a pair of braces(lparen_sym and rparen_sym) and finally an assign_sym followed by an expression. In case the token is a while_sym, control is transferred to expression function. A do_sym token should be followed by a statement_list and an end_sym should conclude the statement if the token generated is if_sym. Control is transferred to expression and when the function returns, a then_sym is checked for followed by invoking the function statement_list. If the next token happens to be else_sym, control is again transferred to function statement list followed by an end_sym.

11.expression list (p,sset,para)

This function checks for an expression or an expression_list followed by a comma_sym and an expression. It invokes function exp_readi(tp,sset) to find out if the identifier or variables used have already been declared.

12.expression(lp,sset,para)

It looks for a simple_expression or a combination of a simple_expression followed by rel_op token(of the type =,<>,>,<,>=,<=) followed by a simple_expression.

13.simple_expression (tp,sset,para)

This function looks for a simple term or an add_op followed by a term (in case of unary operation like +X or -X) or a simple_expression followed by add_op(+,-) and a term.

14. term(tp,sset,para)

This function checks for a factor or a term followed by a mul_op (i.e. mult_sym, dir_sym,divide_sym) followed by a factor.

15.factor(tp,sset,para)

Control is passed on to this function by the function term. It checks for a variable or a constant or an expression within left and right parenthesis. The list of variables and constants are checked in the symbol table and if an entry is not found, an error message is outputted through the function st_error.

Module symbtab.c

The structure of each node of the sy

as follows:

ident_type	--	variable or the proced
ident_name[10]		
label_no	--	the label of the proced
no_raram	--	number of paramete proceudure
para_list[10]		
var_type	--	real or integer variabl
location	--	offset in actuation recor
param	--	this value is 1 if paramet
left	--	node pointer to the left nc
right	--	node pointer to the right

The various functions in this module ar

1) table add():- Increases the symbol_tab arr assigns its value Null, each time it is called.

2)add_ident():- This function returns the nodeptr, which is the address of a struct sym_node. Initially when add_ident() is called, the node pointer p, which is first argument of add_ident, is null. Now it assigns storage of the size of the sym_node and then assigns the address of that storage to p. At the same time it checks for symbol_table[] array. If top of it is found null, it stores the address of the recently allocated storage to symbol_table[top]. Now the ident_name of that structure is assigned to string lex and the left and the right nodes of this structure are assigned null values. Now if it is found that isvar is true i.e. some integer value, it starts increasing the size of the singly linked list, otherwise if isvar is zero; the number of parameters for that sym_node structure is zero and success is assigned one and pointer is returned as the value of it. Next time when add_ident is called, with the pointer p, which is not null now, it checks for string error one, by comparing ident_name field of the structure pointed to by p with lex. If both are the same, success is set to zero. If both are not same, it increases the tree to left side or right side depending on value of condition, by calling add_ident recursively. Therefore add_ident increases the binary tree and singly linked list.

3)generate label():

It generates the label name for each new procedure.

Procedure name starts with "$proc" string and number of procedures is appended to it.

4) Update_type(type,param):

It updates the type of all variables by changing var_type of all sym_nodes which are pointed to by the first field of all nodes of var_list.

5) update_params(p,type):

It updates the no_param field of structure sym_node which is pointed to by p, by increasing it by the number of nodes in var_list.

6) search() :

It searches for the node_ptr p which is passed to it as its argument in binary tree. If p is null, null is returned and if p is the root node of the sub_tree, which is formed as a result of recursive call in search() itself, it returns that p.

7) table_search():

It searches all node pointers in symbol_table[] array which are below a level indicated by t which is an argument of table_search. And that index of symbol_table where node_ptr is found in binary tree is returned as value of level which is a pointer variable of integer type and also argument of table_search(). Otherwise st_error number 2 is returned if number of the node_ptr below level in symbol_table[] is found in binary tree by search() function.

8) free_var_list():

It implies the storage allocated to singly linked list which is pointed by var_list and marks var_list to point to null.

9) free_table(p):

If free, the storage is allocated to the sub_tree of the binary tree which has got p as its root node.

10) print_table:

It prints the ident_name and its location for all node pointers in the subtree of the binary tree which has got p as its root node.

Module type_check.c

1) param_check(p,count,tp):

It checks for number of parameters of sym_node structure pointed to by p with count 1. If it finds counts+1 > p->no_param, then it returns st_error(12), otherwise it checks for para_list[] field for that node_ptr. Now if it finds para_list[] field of p equal to zero, it checks for id_type by comparing tp with argument of param_check with int_type.

2) check_type:

It checks the id_type of two identifiers tp1 and tp2 . It first ensures both tp1 and tp2 are not of err_type. It then checks for tp1 if it is float it sets tp1 to float_type otherwise tp1 is set to integer_type and finally tp1 is returned, otherwise st_error(5) is printed and tp1 is set.

3) assign_check:

It checks first ident_type of sym_node pointed to by pt and if it is equal to 1, st_error(7) is printed; otherwise pt->ident_type = 1, which means pt->var_type will be either 0 or 1. Therefore, tp1 which is a local variable of assign_check() is set to int_type or float_type or err_type depending on whether pt->var_type is 0, 1, or undefined. If pt->var_type is undefined or anything other than 0 or 1, then st_error(3) is printed. Finally, if tp1 is not of err_type, the only assignment error which can occur in the form of st_error(5) is if tp1=int_type and tp2 is float_type i.e. the identifier which is stored in structure sym_nod and pointed to by point is having var_type field equal to 0 and it is being assigned a float_type value in the form tp2.

Module code.c

This module is reponsible for the generation of the final code by the compiler. The code is written in the output file(filename.asm) which has already been opened by the module lex. The module consists of functions:

1) factor_code():

It develops the code factor which is defined in the grammar. factor_code() is called with node_ptr pointer and its field param is checked. If ptr->param is equal to 1, then the sym_node is a variable which is a parameter of a function. Now this parameter is either int_type or float_type. But if ptr->param is not equal to 1 then variable is ordinary local or global variable in some function. Now in first case when variable is parameter, the variable type is compared with int_type and if calling and called integers are same, then it shows that the function is called from within itself i.e. there is a direct recursive call to function. Therefore the code generated is "pushI ptr->location" where ptr->location is off-set for the value of the variable in a frame which is pointed to by FP(a Frame Pointer). That value is taken and stored at the top of stack and "fetchI". This fetchI will make the top of the stack equal to a value of a location pointed to by the value of the top of stack. And if calling <> called i.e. a function called from outside the function. This difference is static level difference. A similar thing is done if idt == float_type. Next, when variable is not parameter, then the only difference is that its value is simply put on the top of the stack.

2) <u>term_divide_code(tp1,tp2)</u>

Both tp1 and tp2 are identifiers of the same or different types.

If tp1 is of int_type and tp2 is of float_type , then output is fltb and divR i.e. it converts tp1 to floating type and then does a real division.

Similar operations are carried out if tp1 is of type floating and tp1 is of integer type.

3) <u>term_mult_code(tp1,tp2)</u>

If both tp1 and tp2 are of the same type, a simple mulI or mulR is outputted. If one of them is of type float, the other is also converted to type float by fltb and than a mulR is outputted.

4) <u>code_exp(tp1,tp2,rel_op)</u>

tp1 and tp2 give the identifier_type and rel_op defines the relational operator between them. If the identifiers are of different types, both of them are converted to real type. Then depending on the rel_op, the assembly code corresponding to operator is outputted.

	integers	reals
lt_sym	ltI	ltR
eq_sym	eqI	eqR
gt_sym	gtI	gtR
lteq_sym	leI	leR
noteq_sym	neI	neR
gteq_sym	geI	geR

5) code_s_exp (tp1,tp2,add_op)

Here again depending on tp1 and tp2 types, addI or subI are outputted in case both operands are integer and addR or subR in case of reals.

6) code_assign(ptr,tp2,calling,called)

Ptr is the pointer to the node containing the identifier tp2. Then if calling = called i.e. the identifier is local to the procedure, it outputs a popI otherwise a popgI if the variable is a global one. Similar operations are carried out in case of reals.

If the variable is of the type parameter, then depending on whether it is real or integer, the following codes are outputted -- popiI and popiR.

Module error.c

This module is invoked when an error is encountered in the lexical analyser or the parser or one of the later modules. It contains the following functions:

1) error(er_no)

This function is envoked by the token generator and depending on the er_no, it outputs a dignostic message telling the user whether the string has a newline character or whether there is an illegal symbol after \\ or if an EOF has been encountered in the comment line and so on.

2) <u>s_error(er_no)</u>

Control is handed over to this function by the lexical analyser if it is found that the system of the input program does not match the grammar of the language. Different er_no values will generate different messages like missing semicolons, colons,program names, begins' and ends' etc.

3) <u>st_error(er_no)</u>

This function takes over after an error has been found during the parsing process and generation of the symbol_table. Error messages include those of undefined variables being used in the programs, mismatch in variables, parameter list mismatch etc.

C O N C L U S I O N

Developing a full compiler is a very complex task requiring intimate knowledge of both the source language as well as the object language. The abstract machine for which this compiler was developed was found to be sufficient for a language like Pascal. But slight difficulty is encountered for implementing floating point constants and variables. Again the code generated is found to be quite compact and takes lesser storage than the corresponding code in any other conventional machine language. The project can be further extended by taking the standard Pascal as an input language and developing its language dependent translator. On the other hand, a machine dependent transaction can be developed to convert the stack-code into the assembly language for any hardware platform. A simple way would be to develop an interpreter which would build a stack in the computer's memory and implement the code directly through user defined procedures like push,pop etc.

›pendix

```
            / * Lex.c */

              /* Lexical analyser */

#include<stdio.h>
#include<ctype.h>
#include<string.h>
#include "GLOBALS.h"
#define LIM 20

char *progname ;
FILE *fp ;
FILE *ofp ;
main(argc,argv)
int argc ;
char *argv[] ;

{
char ch ;
int j = 0;
set st = 0 ;
FILE *efopen();
char outname[LIM] ;
int max = LIM - 5 ;
int pos = 0 ;
char *cp ;

progname = argv[0] ;

if (argc != 2 )
 {
 printf( " usage: %s filename\n", progname ) ;
  return 1 ;

  }
fp = efopen(argv[1],"r" ) ;
cp = strchr(argv[1], '.');
if (cp == NULL )
 {
 printf(" Filename must be of form: name.u \n") ;
return 1 ;
  }
while (( argv[1] + pos) != cp )
 pos++ ;
if (pos <= 0 )
 {
printf("File name must be of the form: name.u \n") ;
return 1 ;
  }
if ( pos <= max)
 {
  strncpy(outname, argv[1],pos) ;
  j = pos;
  }
else
```

```
 {
 strncpy(outname,argv[1],max);


 j = max ;
  }
outname[j] = '.' ;
outname[++j] = 'a' ;
outname[++j] = 's' ;
outname[++j] = 'm' ;
outname[++j] = '\0';
ofp = efopen(outname, "w");
sym = getsym();
program(st = add_prog(st));
fclose(fp) ;
fclose(ofp);
if (is_error)
printf("--- Errors in program\n") ;
else
printf("--- Happy Halt\n");
 }
/*function efopen */
FILE *efopen(file,mode)
     char *file, *mode;
{
FILE *fp, *fopen() ;
    if ((fp = fopen(file, mode)) != NULL )
return fp ;
fprintf(stderr, "%s: Can't open file file %s, mode %s\n",
        progname, file,mode);
exit (1) ;
 }
```

/* Main.c */

```
# include <stdio.h>
# include <ctype.h>
# include <string.h>
# include "externs.h"

static debug = 0 ;
extern FILE *fp ;
char *keyword[] =
  {
   "begin", "div", "do", "else", "end", "if","integer",
   "procedure", "program", "real", "then","var","while'

    };

 token getsym()

  {
 token getsym2();
  token tok ;
  tok = getsym2();
 if (debug) print_tok(tok) ;
 return tok;

   }
   token getsym2()
{
 void error();
  int lower();
int t , last_t, limit, j, i ;
float rem = 1.0;
while (TRUE)
 {
while (( t = getc(fp)) == ' ' ||  t== '\t' || t== '\n'
 if ( t == '\n')

   ++line_no ;
 if ( t == '{' )
  {
    while ((( t = getc(fp)) != '}') && (t != EOF ))
 if ( t == '\n' ) ++line_no ;
  if ( t == EOF )
   {
  error( 5, line_no) ;
     ungetc( t, fp) ;
return error_sym ;
    }
   }
else
if (isdigit(t))

  {
 int_val = t - '0' ;
while ( isdigit(t = getc(fp)))
```

```
 int_val = int_val * 10 + t - '0' ;

 if (( last_t = t) == 46 )
   {
 if (isdigit(t = getc(fp)))
    {
 float_val = int_val ;
 ungetc(t, fp) ;
     }
 else
    {
ungetc(t,fp) ;
 ungetc(last_t,fp) ;
return int_sym ;
     }
    }
else
   {
   ungetc(t, fp);
 return int_sym ;
    }
while (isdigit( t = getc(fp)))
   {
 rem = rem / 10.0 ;
float_val = float_val + ( t - '0' ) * rem ;
    }
ungetc( t, fp) ;
return float_sym ;

   }
else
 if (( last_t = t) >= 60 && t <= 62 )
  {
if (( t = getc(fp)) < 60||  t > 62 )
   {
 ungetc( t , fp); t = 0 ;
    }
switch (t + last_t)
   {
case 60 : return lt_sym ;
 case 61 : return eq_sym ;
 case 62 : return gt_sym ;
 case 121 : return lteq_sym ;
 case 122 : return noteq_sym ;
 case 123 : return gteq_sym ;
    }
   }
else
 if (isalpha(t))
  {
 lex[0] = lower(t) ;
for (limit = 1 ; limit < MAXLEN - 1 &&
   ++limit )

lex[limit] = lower(t) ;
lex[limit] = '\0' ;
ungetc(t, fp) ;
while (isalpha( t = getc(fp))|| isdigit(t)) ;
ungetc(t, fp) ;
for ( j=0 ; j< NO_KEYWORDS && (strcmp(lex,keyword[j])); j++);
switch ( j)
   {
case 0 : return begin_sym ;
case 1 : return div_sym ;
case 2: return do_sym ;
case 3 : return else_sym ;
case 4 : return end_sym ;
case 5 : return if_sym ;
case 6 : return integer_sym ;
case 7: return procedure_sym ;
case 8: return program_sym ;
case 9 : return real_sym ;
case 10 : return then_sym ;
case 11 :  return var_sym ;
case 12 : return while_sym ;
case 13 : return ident_sym ;

    }
   }
else
 if ( t == 39 )
  {
 i = 0 ;

while ( i < MAX_STR_LEN )
   {
 if (( last_t = (t = getc(fp))) == 92 )
 switch ( t = getc(fp))
    {
case 39 : strarray[i] = 92 ;
 strarray [++i] = 39 ; ++i ; break ;
 case 92 : strarray[i] = 92 ;
 strarray[++i] = 92; ++i ; break ;
 case 110 : strarray[i] = 92 ;
   strarray[++i] = 110 ;
   ++i
    ; break ;

 default : strarray[i] = 92 ; strarray[++i] = t ;
           error( 3, line_no) ; ++i ; break ;
     }
 else
 switch(t)
    {
case 39 : strarray[i] = '\0'  ;
          return string_sym ;
 case 10 : ++line_no ;
             error (1, line_no) ;
  default : strarray[i] = t ; ++i ;

    }
   }

 if ( i >= MAX_STR_LEN) error( 1, line_no) ;
 while (( t = getc(fp)) != 39 );
   }
 else
  if ( t == EOF ) return eof_sym ;
 else
  {
 switch (t)
   {
 case 40 : return  lparen_sym ;
 case 41 : return  rparen_sym  ;
 case 42  : return   mult_sym  ;
 case 43   : return   plus_sym ;
 case 44 : return   comma_sym  ;
 case 45 : return   minus_sym  ;
 case 46 : return   period_sym ;
 case 47 : return   divide_sym ;
 case 58   : if (( t = getc(fp)) == 61 )
            return   assign_sym ;
 else
    {
      ungetc( t, fp ) ;
      return colon_sym ;
     }
  case 59 : return semicolon_sym ;
 default : error( 4 , line_no ) ;
     printf( "sym %d\n", t ) ; break ;
    }
   }
  }
 }

 int lower(c)
int c ;

{
if ( c >= 'A' && c <= 'Z' )
 return c + 'a' - 'A' ;
else
 return c ;
 }
```

```
                        /* Code.c */
                        /* Code generator */

# include <stdio.h>
# include "externs.h"
extern FILE *ofp ;

void factor_code(ptr, calling, called, idt )
 node_ptr ptr ;
 int calling ;
 int called ;
 id_type idt ;

{
 if (ptr->param == 1 )
 {
 if ( idt == int_type )
  if (calling == called)
  {

 fprintf(ofp, "      pushI %d\n",  ptr->location );
 fprintf(ofp, "      fetchI\n") ;
   }

else
  {
  fprintf( ofp,"      pushI %d, %d\n",
                 calling-called, ptr->location);
  fprintf(ofp, "      fetchI\n" );

   }
 else
 if (idt == float_type)
 if (calling == called )
  {
 fprintf(ofp, "      pushI %d\n", calling-called,
         ptr->location);
   }
  }
 else
 {
       if (idt == int_type )
      if ( calling == called )
     fprintf(ofp, "      pushI %d\n", ptr->location);
else
 fprintf(ofp,"       pushgI %d, %d\n", calling-called,
         ptr->location) ;
 else
 if (idt == float_type)
 if (calling == called)
 fprintf(ofp, "      pushR %d\n", ptr->location);
 else
 fprintf(ofp, "      pushgR %d\n", calling-called,
 ptr->location);
  }
  }
 void factor_para_code(ptr, calling, called, idt)
  node_ptr ptr ;
 int calling ;
 int called ;
 id_type  idt ;
 {
 if ( ptr->param == 1 )
  {
 if (idt == int_type )
 if (calling == called )
 fprintf(ofp,"      pushI %d\n", ptr->location );
 else
    fprintf(ofp,"      pushgI %d, %d\n ", calling-called,
            ptr->location) ;
 else
 if ( idt == float_type )
       if ( calling == called )
 fprintf(ofp, "      pushR %d\n", ptr->location) ;
 else
    fprintf(ofp, "      pushgR %d, %d\n", calling-called,
            ptr->location );

   }
 else

  {
 if (calling == called )
 fprintf( ofp,"      pusha  %d\n",ptr->location);
  else
fprintf( ofp,"      pushga  %d, %d\n",calling-called,
        ptr->location);
   }
  }
void term_divide_code(tp1, tp2)
 id_type tp1 ;
 id_type tp2 ;
 {
   if ( tp1 == int_type )
   {
       if ( tp2 == float_type )
    {
 fprintf(ofp, "      fltb\n");
fprintf( ofp,"      divR\n");

     }
else
if (tp2 == int_type)
```

```
    {

fprintf( ofp,"      flt\n");

fprintf( ofp,"      divR\n ");
     }
   else
   if (tp2 == float_type )
      fprintf( ofp , "      divR\n");
   }
 void term_mult_code(tp1, tp2)
 id_type tp1 ;
 id_type tp2 ;
  {
      if ( tp1 == int_type)
    {
     if ( tp2 == float_type )
     {
 fprintf( ofp, "      fltb\n" ) ;
  fprintf( ofp, "      mulR\n" ) ;
      }
else
 if ( tp2 == int_type )
   fprintf( ofp, "      mulI\n ");

      }
 else
 if (tp1==float_type)
  if(tp2==int_type) {
    fprintf(ofp,"      flt\n");
    fprintf(ofp,"      mulR\n");
   }
else
   if (tp2 == float_type)
      fprintf(ofp, "      mulR\n");}
 void code_exp(tp1, tp2, rel_op)
 id_type tp1 ;
 id_type  tp2 ;
 token     rel_op ;
  {
if ( tp1 == int_type )
   {
 if ( tp2 == float_type )
 fprintf( ofp, "      fltb\n" ) ;
    }
 else
 if ( tp1 == float_type )
if ( tp2 == int_type )
 fprintf( ofp, "      flt\n");
    if (( tp1 == int_type) && ( tp2 == int_type ))
   {
 switch (rel_op)
    {
```

```
case lt_sym : fprintf( ofp,"      ltI\n"); break ;
case eq_sym : fprintf( ofp,"      eqI\n" ); break ;
case gt_sym : fprintf( ofp,"      gtI\n " ); break ;
case lteq_sym : fprintf( ofp,"      leI\n" ); break ;
case noteq_sym : fprintf( ofp,"      neI\n" ); break ;
case gteq_sym : fprintf( ofp,"      geI\n" ) ; break ;
 .  }
   }
 else
switch (rel_op)
  {
case lt_sym : fprintf( ofp,"      ltR\n"); break ;
case eq_sym : fprintf( ofp,"      eqR\n" ); break ;
case gt_sym : fprintf( ofp,"      gtR\n " ); break ;
case lteq_sym : fprintf( ofp,"      leR\n" ); break ;
case noteq_sym : fprintf( ofp,"      neR\n" ); break ;
case gteq_sym : fprintf( ofp,"      geR\n" ) ; break ;

   }
  }
void code_s_exp( tp1, tp2, add_op )
id_type  tp1 ;
id_type   tp2 ;
token add_op ;
  {
  if ( tp1 == int_type ) {
if ( tp2 == int_type )
     switch ( add_op) {
 case plus_sym : fprintf( ofp,"    addI\n"); break ;
 case minus_sym : fprintf( ofp,"    subI\n"); break;
   }
}
else
   {
if ( tp1 == float_type )
if (tp2 == int_type )
  fprintf(ofp, "     flt\n");
if ( tp1 == int_type )
fprintf(ofp, "     fltb\n");
if ( tp2 == float_type )
fprintf( ofp,"     fltb\n");
switch(add_op)
{
   case plus_sym : if (( tp1 == float_type )||
                                        ( tp2 == float_type))
                    fprintf(ofp,"     addR\n");
               else
             if  (( tp1 == int_type) && ( tp2 == int_type))
                  fprintf(ofp, "     addI \n "); break ;
   case minus_sym  : if (( tp1 == float_type)||
                                         (tp2 == float_type))
                     fprintf(ofp, "     subR\n ");
                    else
                    if (( tp1 == int_type ) &&
                                       ( tp2 == int_type ))
                    fprintf(ofp,"      subI \n") ; break ;
                     }
                }
            }

        void code_iflhs_para(ptr, calling, called)
        node_ptr ptr ;
       int calling ;
       int called ;
      {
        id_type tp1 ;
         if ( ptr -> var_type)  tp1 = float_type ;
           else tp1 = int_type ;
    if ( ptr ->param == 1 )
   {
        if ( tp1 == int_type )
      if (called == calling )
      fprintf ( ofp,"      pushI %d\n", ptr->location);
        else
         fprintf(ofp,"      pushgI %d\n",calling-called,
                 ptr->location);
                }
            }
    void code_assign(ptr, tp2, calling, called)
    node_ptr ptr ;
    id_type  tp2 ;
    int calling ;
   int called ;
   {
      id_type tp1 ;
      if ( ptr->var_type) tp1 = float_type ;
      else tp1 = int_type ;
      if (ptr->param  != 1 )
    {
     if ( tp1 == int_type ){
     if ( tp2 == int_type)
         if (called == calling )
    fprintf(ofp,"      popI %d\n",ptr->location);
   else
     fprintf(ofp,"     popgI %d\n",calling-called,ptr->location);
        }
       else
       {
       if ( tp1 == float_type )
       if ( tp2 == float_type )
      if ( called == calling )
       fprintf(ofp, "      popR %d\n", ptr->location);
        else
       fprintf(ofp,"      popgR %d, %d\n", calling-called,
               ptr->location);
        else
              if ( tp2 == int_type)
 {
      fprintf( ofp,"      flt " ) ;
         if ( called == calling )
  fprintf( ofp,"      popR%d\n", ptr->location ) ;
 else
          fprintf( ofp,"      popgR  %d %d\n", calling-called
                                              , ptr->location ) ;
                         }
      }
       }

else

     {
if ( ptr->param == 1 )
   if ( tp1 == int_type ){
        if ( tp2 == int_type )
    fprintf( ofp , "      popiI\n");
}
   else
        {
   if ( tp1 == float_type )
   if ( tp2 == float_type )
    fprintf( ofp,"      popiR\n " );
       else
    if ( tp1 == int_type )
  {
fprintf( ofp,"      flt" ) ;
fprintf( ofp,"      popiR\n " ) ;
                         }
                    }
                }
             }


   int strleng(st)
  char *st ;
 {
int len ;
for ( len = 0 ; *st != '0'; st++)
    {
       if ( *st == '\\')
   ++st ;
len++ ;
}
  return len ;
   }
void code_std_proc(pt, tp, vt, vlevel )
node_ptr pt ;
```

```
id_type  tp  ;
node_ptr  vt ;
 int  vlevel ;
{

 if (( strcmp(pt->ident_name, "writei" )) == 0 )
fprintf(ofp,"      outI\n " ) ;
     else
 if (( strcmp(pt->ident_name, "writer" )) == 0 )
{
  if (tp == float_type)
    fprintf( ofp,"      outR\n" ) ;
   else
      if (tp == int_type){
    fprintf(ofp,"       flt\n");
    fprintf( ofp,"      outR\n" ) ;

               }
      }
  else
if ((strcmp (pt->ident_name, "writetxt" )) == 0 )
  {
 fprintf( ofp,"      pushcI   %d\n",strleng(strarray ) ) ;
 fprintf( ofp,"      msg \ '%s' \n",  strarray ) ;
     }
    else
if (( strcmp (pt->ident_name, "readi" )) == 0)
       {
if ( tp == int_type )
{
fprintf( ofp,"      inpI\n" ) ;
       if ( top == vlevel )
     fprintf( ofp,"      popI %d\n", vt->location) ;
     else
     fprintf( ofp,"      popgI  %d,%d\n",  top-vlevel
                            , vt->location ) ;
            }
       }
else
     if (( strcmp(pt->ident_name, "readr" )) == 0 )
   {
   if (tp == float_type )
    {
        fprintf(ofp, "      inpR\n" );
    if ( top == vlevel )
     fprintf (ofp, "      popR %d\n", vt->location ) ;
    else
  fprintf (ofp, "      popgR %d, %d\n", top-vlevel,
           vt->location);

       }
      }
  }
```

```
                         /* Par.c */
                         /* Parser */

#include <stdio.h>
#include "externs.h"

extern FILE *ofp;
static int success;
static node_ptr ptr = NULL;
static node_ptr proc = NULL ;
static debug = 0;
node_ptr add_ident(), search();

void block(),decls(),var_decls(),var_decl(),proc_decls(),
 proc_decl(),
proc_header(), parameter_list(),
 parameter_group(), statement_list(),
statement(), expression_list(), expression(),
  simple_expression(),
term(),factor(), identifier_list(),
expression_std_proc(), code_std_proc(), exp_readi();
void program(sset)
 set sset;
{
 set tset;
  int no_var=0;
  int no_para = 0 ;
  node_ptr *dummy = (node_ptr *) malloc( sizeof(node_ptr) ) ;

if (debug) printf( "-In Program \n" );
   table_add();
   install_procs();

if (sym == program_sym)
    sym = getsym();
else
{
s_error(1) ;
sym = syncr(sset);
}
 if (sym == ident_sym)
  sym = getsym();
else
{
s error(2);
```

```
 tset = add_block(sset);
block (tset, &no_var, &no_para, dummy);
 if (sym == period_sym)
     sym = getsym();
else
{
s_error(4);
sym = syncr(sset = add_set(eof_sym, sset));
}
 if ( debug) printf( "-Out Program\n ") ;
}

void block(sset, no_var, no_para, proc_ptr)
set sset;
int *no_var;
int *no_para;
node_ptr *proc_ptr ;
{
set tset ;
 int int_temp ;

if (debug) printf(" -In Block\n" );
     int_temp = *no_para ;
      no_var = &int_temp ;
      decls(sset, no_var, no_para ) ;

if (* proc_ptr != NULL)
{
 fprintf(ofp, "   %s%d\n",proc_label, (*proc_ptr)-> label_no);
  fprintf(ofp, "  alloc %d\n", *no_var);
}
else
{
fprintf(ofp," main\n " ) ;
 fprintf(ofp, "     enter 0 \n " );
  fprintf(ofp, "     alloc %d\n", *no_var ) ;
}
if ( sym == begin_sym )
  sym = getsym();
else
{
s_error(6);
sym = syncr(sset);
}
 tset = add_st_list (sset);
 statement_list(tset);
 if (sym == end_sym)
{
 free_table(symbol_table[top]) ;
 top--;

fprintf(ofp ,"   return\n");
sym = getsym();
}
else
{
s_error(5) ;
sym = syncr(sset);
}
if (debug) printf( "-Out Block\n");
}
void decls(sset, no_var, no_para)
 set sset;
int *no_var;
int *no_para ;

{
set tset ;
int var = 0 , para = 0 ;
if (debug) printf("-In Decls\n");
while (sym != begin_sym )
{
if (sym == var_sym )
{
var_decls(sset = add_decls(sset), no_var);
var = *no_var - *no_para; para = *no_para ;
}
else
if (sym == procedure_sym)
{
var = *no_var - *no_para ;para = *no_para ;
proc_decls(sset = add_decls(sset), no_para);
}
else
 sym = syncr(sset = add_decls(sset));
}
*no_var = var ; *no_para = para ;
 if (debug) printf("-Out Decls\n");
}
void var_decls(sset, no_var)
set sset;
 int *no_var ;
{
if (debug) printf("-_In Var_decls\n");

sym = getsym();

var_decl(sset, no_var);
while (sym == ident_sym)
var_decl(sset , no_var);
if (debug) printf("-Out Var_decls\n");
}
void var_decl(sset, no_var)
set sset;
int *no_var;
```

```
{
set tset;
if (debug) printf("-In var_decl\n");
tset = add_id_list(sset);
identifier_list(tset, no_var);
if( sym == colon_sym )
 sym = getsym();
else
{
s_error(8);
sym = syncr(sset);
}
if (( sym == integer_sym) || (sym ==real_sym))
{
if (sym == integer_sym)
 update_type(0, 0);
else
update_type(1,0 );
sym = getsym();
}
else
{
s_error(7);
sym = syncr(sset);
}

free_var_list();
if (sym == semicolon_sym)
sym = getsym();
else
{
s_error(3) ;
sym = syncr(sset);
}
if (debug) printf( "-Out Var_decl\n");
}
void proc_decls(sset, no_para)
set sset;
```

```
if (debug) printf("-In Proc_decl\n");
proc_header(sset,no_para, &proc_ptr);
block(sset,dummy, no_para,&proc_ptr);
if (sym == semicolon_sym )
sym = getsym();
  else
 {
s_error(3);
sym = syncr(sset);
}
if (debug) printf( "-Out Proc_decl\n");
}
void proc_header(sset, no_para, proc_ptr)
set sset ;
int *no_para ;
 node_ptr *proc_ptr;
{
set tset ;
 int level = 0 ;
node_ptr temptr = (node_ptr) malloc(sizeof(sym_node));

if (debug) printf("-In proc_header\n");
if (( sym = getsym() ) == ident_sym )
{
 add_ident(symbol_table[top],success,0,0);

proc = table_search(top, &level);

temptr = proc;
*proc_ptr = temptr ;
proc-> label_no = proc_no++;
sym = getsym();

}
else
{
s_error (11);

sym = syncr(sset) ;
}
*no_para = 0 ;
table_add();
if (sym == lparen_sym)
{
sym = getsym();
tset= add_p_list(sset);
parameter_list(tset, no_para);
if (sym == rparen_sym )
sym = getsym();
else
{
s_error(9);
sym = syncr(sset);
```

```
}
if (sym == semicolon_sym)
sym = getsym();
else
{
s_error(3);
sym = syncr(sset);
}
}
else
 if (sym == semicolon_sym)
sym= getsym();
else {
s_error(10);
sym = syncr(sset);
}
if (debug) printf("-Out proc_header\n");
}

void parameter_list(sset, no_para)
set sset ;
int *no_para;
{
if (debug) printf("-In Parameter_list\n");
parameter_group(sset,no_para);
while (sym == semicolon_sym)
{
sym = getsym();
parameter_group(sset, no_para);
}
if (debug) printf("-Out Parameter_list\n");
}
void parameter_group(sset, no_para)
set sset;
int *no_para;
{
set tset;
if (debug)  printf("-Out Parameter_group\n");
tset = add_id_list(sset);
identifier_list (tset, no_para);
if (sym == colon_sym)
sym = getsym();
else

{
s_error(8);
sym = syncr(sset) ;
}
if (( sym == real_sym)||(sym == integer_sym))
{
if (sym == integer_sym)
{
update_type(0, 1 );
```

```
update_params(proc, 0 );
}
else
{
update_type(1,1);
update_params(proc,1);
}
sym = getsym();
}
else
{
s_error(7);
sym = syncr(sset);
}
free_var_list();
if (debug) printf( " -Out Parameter_group\n");
}
 void statement_list(sset)
 set sset ;
{
if (debug) printf("-In statement_list\n");
statement(sset);
while ((sym == semicolon_sym)||(sym == if_sym)
 ||(sym == while_sym)||(sym == ident_sym))
{
if (sym == semicolon_sym)
 sym = getsym();
else
  s_error(16) ;
  statement(sset);

}
if (debug) printf("-Out Statement_list\n");
}
void statement(sset)
set sset;
{
  node_ptr pt ;
  id_type tp ;
  set  tset ;
  int level ;
if (debug) printf("-In Statement\n");
if (sym == ident_sym)
{
pt = table_search(top, &level);
if (( sym = getsym()) == assign_sym)
{
 code_iflhs_para(pt,top,level);
 sym = getsym();
 expression( &tp,tset= add_exp_assign(sset),0);
 if (pt !=NULL)
  assign_check(pt, tp);
 else
 st_error(14);
 code_assign(pt, tp, top, level );
}
else
 if (sym == lparen_sym)
  {
   if (pt != NULL )
     if (pt-> ident_type == 0)
      {
        fprintf( ofp, "     enter %d\n", top- level+1 ) ;
        sym = getsym();
        expression_list(pt, tset = add_exp_stat(sset), 1 ) ;
        if ( sym == rparen_sym)
         {
          sym = getsym();
           fprintf(ofp, "   call %s%d, %d\n", proc_label,
                   pt->label_no,pt->no_param);

         }
         else
            {
             s_error(9);
             sym = syncr(tset );
            }
         }
      else
        if (pt->ident_type == 3 )
        {
          id_type tp;
          node_ptr vt;
          int vlevel ;

          sym = getsym();
          vt = table_search(top, &vlevel);
          expression_std_proc(pt, tset = add_exp_stat(sset),
                                       &tp);
          if (sym == rparen_sym)
          sym = getsym();
          code_std_proc( pt, tp, vt, vlevel);
        }
     else
          {
           st_error(9);
           sym = syncr(tset = add_not_proc( tset));
          }
       else
          {
           st_error(14) ;
           sym = syncr(tset) ;
          }
        }
     else
        if ( sym == semicolon_sym)
         if ( pt !=NULL )
```

```
            if ( pt -> ident_type == 0 )
              if ( pt-> no_param == 0 ) {
                     fprintf( ofp, "   enter %d\n",
                                           top- level +1 );
                     fprintf(ofp, "    call %s%d, %d\n" ,
                             proc_label,pt->label_no,
                                            pt->no_param );

              }
else
   st_error(15) ;
 else
  st_error(9);
   else
        st_error(14);
  else
     {
       s_error(16);
          sym = syncr(sset);
 }
}
else
  if ( sym == while_sym)
  {
 int while_no ;
while_no = while_count++ ;
fprintf( ofp, " %s%d\n", while_lab, while_no ) ;
 sym = getsym();
 expression ( &tp, tset = add_exp_while(sset), 0);
 if ( tp != bool_type)
 {
   st_error(6) ;
sym = syncr( tset = rm_set(ident_sym,tset));
}
    if ( sym == do_sym)
   sym= getsym();
 else
{
    s_error(12);
 sym = syncr( tset = add_set(ident_sym, tset));
  }
statement_list(tset = add_st_list(sset));
fprintf(ofp, "    jump %s%d\n", while_lab, while_no);
fprintf(ofp, " %s%d\n" ,w_end_lab, while_no );
if (sym == end_sym )
sym = getsym() ;
else
{
 s_error(5) ;
   sym = syncr(sset) ;
 }
}
else
```

```
   if   ( sym == if_sym)
   {
   int if_no ;
  if_no = if_count++ ;
sym = getsym();
expression( &tp, tset = add_exp_if(sset), 0 ) ;
 fprintf ( ofp, "   jumpz %s%d\n", else_lab, if_no) ;
    if (tp != bool_type)
     {
  st_error(6) ;
  sym = syncr(tset = rm_set(ident_sym, tset));
 }
  if ( sym == then_sym )
    sym = getsym();
  else
 {
   s_error(15);
sym = syncr(tset = add_set(ident_sym, tset));
 }
statement_list (tset = add_st_list(sset) );
if (sym == else_sym)
     {
       fprintf(ofp, "-     jump %s%D\n", if_end_lab,if_no);
fprintf(ofp, " %s%d\n",else_lab, if_no);
sym = getsym() ;
 statement_list(tset) ;
fprintf (ofp, "%s%d\n", if_end_lab, if_no);
if (sym == end_sym)
sym = getsym();
else {s_error(5); sym = syncr(sset);}
}
else
 if (sym == end_sym )
 {
 fprintf( ofp, "%s%d\n", else_lab, if_no) ;
 sym = getsym();
}
 else
{
   s_error(5);
 sym = syncr(sset);
}
}
if (debug) printf("-Out Statement\n" ) ;
}
void expression_std_proc(pt, sset,tp)
node_ptr pt ;
 set sset;
id_type *tp ;
 {
 void param_check(), expression(),exp_readi();
int para = 0 , count = 0 ;
set tset ;
```

```
 if (debug) printf("-In Exp_sdt_proc\n");
if ((strcmp(pt->ident_name, "readi " ) == 0 )||
  (strcmp(pt->ident_name, "readr" ) == 0 ))
 {tset = add_e_lst(sset);
 exp_readi (tp,tset);}
 else
  expression(tp,tset = add_e_lst(sset), para );
 param_check(pt,count, *tp) ;
 if (debug) printf("-Out Exp_sdt_proc\n");
}
 void exp_readi(tp, sset)
 id_type *tp ;
  set sset ;
 {
 node_ptr p ;
int level ;
if ( sym == ident_sym)
  {
    p = table_search(top, &level) ;
 if ( p != NULL )
 {
       if ( p->ident_type == 1 )
 {
 sym = getsym() ;
 switch ( p->var_type)
 {
  case 0 : *tp = int_type ; break ;
  case 1 : *tp = float_type ; break ;
default : st_error(3) ; *tp = err_type ;
}
}
 else
 {
 st_error(2) ;
 syncr(sset);
}
}
   else
    {
st_error(15) ;
syncr(sset);
}
}
}

 void expression_list (p, sset, para)
 node_ptr p;
 set sset ;
 int para;
{
void param_check() , expression();
int count = 0 ;
 id_type tp ;
 set tset ;

   if (debug) printf("-In Exp_list\n") ;
expression(&tp, tset = add_e_lst(sset),para);
param_check( p, count, tp );
while (sym == comma_sym)
{
 sym = getsym();
expression(&tp, tset, para);
 ++count ;
param_check(p, count, tp);
}
if (count+1 != p-> no_param) st_error(11) ;
 if (debug) printf("-Out Exp_list\n");
}
 void expression (tp, sset, para)
 id_type *tp ;
 set sset ;
 int para ;
  {
id_type tp1,tp2, tp3 ;
 short rel_flag =0 ;
set tset ;
token rel_op ;
if (debug) printf(" -In Exp\n");
simple_expression(&tp1, tset= add_s_exp(sset),para);
while ((( rel_op = sym ) >= lt_sym) && (sym<= gteq_sym ))

   {
     rel_flag = 1 ;
 sym = getsym();
simple_expression(&tp2, tset,para) ;
code_exp(tp1, tp2  ) ;
tp1 = check_type(tp1, tp2, rel_op );
 if ( tp1 != err_type) tp3 = bool_type ; else tp3 = tp1 ;
}
if (rel_flag) *tp = tp3 ;
else
      *tp = tp1 ;
if (debug )  printf( " -Out Exp\n");
}
void simple_expression(tp, sset,para)
 id_type *tp ;
set sset;
int para ;
{
 id_type tp1, tp2 ;
  set tset ;
token add_op;
if (debug) printf("-IN S_exp\n");
if ((( add_op = sym ) == plus_sym )||(sym == minus_sym))
sym = getsym();
```

```
 term (&tp1, tset = add_term(sset), para) ;
 if ( add_op == minus_sym )
 if (tp1 == int_type )
 fprintf(ofp, "      negI\n" );
else
 fprintf(ofp, "     negR\n");
while ((( add_op = sym ) == plus_sym)||(sym ==  minus_sym))
 {
 sym = getsym();
 term (&tp2, tset, para ) ;
code_s_exp(tp1, tp2, add_op);
tp1 = check_type (tp1, tp2 );
}
*tp = tp1 ;
if (debug) printf("-Out S_exp\n");
}
void term (tp, sset,para )
 id_type *tp ;
set sset;
int para ;
 {
token op_sym ;
id_type tp1, tp2;
set tset ;
if (debug) printf ("-In Term \n ");
 factor (&tp1, tset = add_factor(sset), para );
while ((( op_sym = sym ) == mult_sym)||(sym == div_sym )
                                        ||(sym== divide_sym))

 {
  sym = getsym();
 factor (&tp2, tset, para );
 switch ( op_sym)
     {
      case mult_sym  : term_mult_code(tp1,tp2);
                        tp1 = check_type(tp1,tp2); break ;

      case div_sym   : if ((tp1 == int_type) &&
                                          (tp2 == int_type))
             {
                   tp1 = int_type ;
                     fprintf(ofp,"    div\n") ;
             }
                 else
                   {
                   tp1 = err_type ;st_error(5) ;

                          }          break;
      case divide_sym : if (( tp1 == int_type ||
                                 tp1 == float_type)
                       && ( tp2 == int_type ||
                                tp2 == float_type ))
                     {
```

```
   term_divide_code(tp1, tp2) ;
                         tp1 = float_type ;
                         }
              else
                   {
                    tp1 = err_type ; st_error(5) ;
                                    }
                                       break ;

                 }
}
*tp = tp1;
 if (debug) printf("- Out Term\n");
}
void factor (tp, sset,para)
  id_type *tp ;
    set sset ;
    int para ;
{
  node_ptr p;
 int level ;

  if (debug) printf("-In Factor \n ");

if (sym == ident_sym)

{
        p= table_search(top , &level);

if (p != NULL )
  {
        if ( p->ident_type == 1 )

{
  switch ( p-> var_type)
  {
     case 0 : *tp = int_type ;
                if (para)
                factor_para_code(p,top,level, int_type );
               else
                factor_code( p, top,level,int_type) ;
               break ;

     case 1 : *tp = float_type ;
                   if (para)
                   factor_para_code(p, top, level,int_type);
                   else
                     factor_code(p, top,level, float_type) ;
                     break ;
                   default : st_error(3) ;
                      *tp = err_type ;
         }
       }
```

```
        }
      else
        *tp = err_type ;
      sym = getsym();
       }
    else
    if (sym == string_sym)
          {
             *tp = err_type ;
             sym = getsym();
     }
    else
       if (sym == int_sym )
      {
       *tp = int_type ;
        fprintf(ofp,"       pushcI  %d\n", int_val );
        sym = getsym() ;
        }
    else
   if (sym == float_sym )
     {
         *tp = float_type ;
    fprintf(ofp, "      pushcR %f\n ", float_val ) ;

    sym = getsym();
   }
   else
      if (sym == lparen_sym)
       {
         sym = getsym();
    expression(tp, sset,para) ;
    if ( sym == rparen_sym)
      sym = getsym() ;
    else
    {
     s_error(9) ;
    sym = syncr(sset) ;
       }

   }
    else
   { s_error(13) ;
    *tp = err_type ;
   sym = syncr(sset) ;
   }
    if (debug) printf("-Out Factor \n " );
   }
    void identifier_list(sset, no_var)
    set sset ;
   int *no_var ;
   {
    if (debug) printf("- In Ident_list\n");
    if (sym == ident_sym)
```

```
{
    add_ident(symbol_table[top], success, 1, ++(*no_var));
     sym = getsym();
  }

 else
 {
s_error(14);
 sym = syncr(sset) ;
}
while (sym == comma_sym)
if (( sym = getsym()) == ident_sym)
{
 add_ident(symbol_table[top],success, 1, ++(*no_var)) ;
  sym = getsym();
 }

 else
  {
 s_error(14) ;
sym = syncr(sset) ;
 }
if (sym != colon_sym) sym = syncr(sset) ;
if (debug) printf("-Out Ident_list\n " );

      }
```

```
                    /* Error.c */
                    /* Error handler */

# include <stdio.h>
 # include "externs.h"

 void error(er_no)
 int er_no;

 {
  printf ("%d", er_no);
  }

 void s_error ( er_no)
 int er_no ;
 {
     is_error = 1 ;
 switch (er_no )
  {
 case 1  : printf(" Program symbol  expected at line %d\n",
                   line_no) ; break;
 case 2  : printf("Program name  expected at line %d\n",
                   line_no) ; break;
 case 3  : printf("  Semicolon expected at line %d\n",
                   line_no) ; break;
 case 4  : printf("Period  expected at line %d\n", line_no) ;
                              break;
 case  5 : printf("End  expected at line %d\n", line_no) ;
                              break;
 case 6  : printf("Begin  expected at line %d\n", line_no) ;
                              break;
 case 7  : printf("Type  expected at line %d\n", line_no) ;
                              break;
 case 8  : printf("Colon  expected at line %d\n", line_no) ;
                              break;
 case 9 : printf("Right parenthesis  expected at line %d\n",
                   line_no) ; break;
 case 10 : printf("Left parenthesis  expected at line %d\n",
                   line_no) ; break;
 case 11  : printf("Procedure expected at line %d\n",
                   line_no) ; break;
 case 12 : printf("Do  expected at line %d\n", line_no) ;
                              break;
 case 13 : printf("Error in Factor at line %d\n", line_no) ;
                              break;
 case 14  : printf("Identifier  expected at line %d\n",
                    line_no) ; break;
 case 15  : printf("Then  expected at line %d\n", line_no) ;
                               break;
 case 16  : printf("Error in the  statement at line %d\n",
                    line_no) ; break;

   }
  }
 void st_error(er_no)
 int er_no ;
```

```
    {
 is_error = 1 ;
 switch(er_no)
     {
 case 1 : printf("Redeclaration of variable %s at line %d\n",
                  lex, line_no);break ;
case 2 : printf("Identifier %s undeclared in  line %d\n",
                  lex, line_no);break ;
case 3 : printf("Type of variable %s undefined  in line %d\n",
                  lex, line_no);break ;
case 4 : printf("Identifier %s not declared as a as a variable
                 in line %d\n", lex, line_no);break ;
case 5 : printf("Type mismatch  in line %d\n",  line_no);
        break ;
case 6 : printf("Boolean expression expexted  in line %d\n",
                   line_no); break ;
case 7 : printf("Variable expected on the lefthand side of
                assignment,");
         printf (" at  line %d\n", line_no);
                                break ;
case 8 : printf("Type mismatch in assignment statement in
                line %d\n", line_no);
                                break ;
case 9 : printf("Procedure name expected  in line %d\n",
                line_no);break ;
case 10 : printf("Type mismatch in parameter list of procedure
                   at");
         printf("  in line %d\n", line_no);
                                 break ;
case 11  : printf(" Mismatch in the no of parameters in
                   procedure at");
          printf(" in line %d\n", line_no);
                                 break ;
case 12 : printf("Too many parameters in procedure  in line
                   %d\n", line_no);break ;
case 13  : printf("String expected in the argument at line
                   %d\n", line_no);break ;
case 14 : printf("Identifier undeclared  in line %d\n",
                   line_no);break ;
case 15 : printf("Parameter expected   in line %d\n",
                   line_no);break ;
case 16 : printf("Identifier expected   in line %d\n",
                   line_no);break ;
       }
      }
 void set_error(k)
     int k ;
    {
  is_error = 1 ;
  switch (k)
     {
  case 1: printf("Can't put element in the set\n"); break;
      }
     }

   void print_tok(p_sym)
  token p_sym ;
  {
    switch (p_sym)

   {
 case  int_sym  : printf(" int_sym %d\n int_val " ); break;

 case  float_sym : printf(" float_sym %f\n float_val" ); break;
 case  lt_sym  : printf(" lt_sym\n" ); break;
 case  eq_sym  : printf(" eq_sym\n" ); break;
 case  gt_sym  : printf(" gt_sym\n" ); break;
 case  lteq_sym  : printf(" lteq_sym\n" ); break;
 case  noteq_sym  : printf(" noteq_sym\n" ); break;
 case  gteq_sym  : printf(" gteq_sym\n" ); break;
 case  lparen_sym  : printf(" lparen_sym\n" ); break;
 case  rparen_sym  : printf(" rparen_sym\n" ); break;
 case  mult_sym  : printf(" mult_sym\n" ); break;
 case  plus_sym  : printf(" plus_sym\n" ); break;
 case  comma_sym  : printf(" comma_sym\n" ); break;
 case  semicolon_sym  : printf(" semicolon_sym\n" ); break;
 case  begin_sym  : printf(" begin_sym\n" ); break;
 case  do_sym  : printf(" do_sym\n" ); break;
 case  div_sym  : printf(" div_sym\n" ); break;
 case  else_sym  : printf(" else_sym\n" ); break;
 case  end_sym  : printf(" end_sym\n" ); break;
 case  if_sym  : printf(" if_sym\n" ); break;
 case  integer_sym  : printf(" integer_sym\n" ); break;
 case  procedure_sym  : printf(" procedureplus_sym\n" ); break;
 case  program_sym  : printf(" program_sym\n" ); break;
 case  real_sym  : printf(" real_sym\n" ); break;
 case  then_sym  : printf(" then_sym\n" ); break;
 case  var_sym   : printf ("var_sym\n"); break ;
 case while_sym : printf( "while_sym\n"); break ;
 case ident_sym : printf( "ident_sym\n"); break ;
 case string_sym : printf( " string_sym %s\n " , strarray );
                                    break ;
  case eof_sym :     printf( " eof_sym\n" ) ;
    }
   }
```

/* **Sets.c** */

```
# include <stdio.h>
# include "externs.h"

static debug = 0 ;
set setA = 0 ;

set  add_set (elem, setvar)
  token elem;
 set setvar;
{
  if (elem < 32)
 return (setvar = setvar | ( 1 << elem ));
else set_error(1);
 }
set rm_set (elem, setvar)
 token elem ;
  set setvar;
{
 if (elem < 32)
  return (setvar & ( 1 << elem));
 else set_error (1);
 }
  int is_memb(elem, setvar)
 set setvar;
{
if (elem < 32)
 return ( setvar & ( 1 << elem));
  else  return 0;
 }
void print_set(setvar)
 set setvar;
{
 int i;
 printf ("-In print_set \n");
 for ( i = 0 ; i < 32 ; i++ )
  if ( is_memb (i, setvar ))
   print_tok (i);
 printf("-out print_set\n");
 }

 token syncr(setvar)
 set setvar;
{
token tok;
tok = sym;
if (debug) printf("-In syncr\n");
if (is_memb(sym, setvar))
 return tok ;
 else

 while( !is_memb(tok, setvar))
 tok = getsym();
if (debug) printf("-out syncr\n");
```

```
  return tok;
   }

 set add_prog(setvar) /* called from main */
      set setvar ;
 {

 setvar = add_set (procedure_sym, setvar);
 setvar = add_set(var_sym, setvar);
 setvar = add_set(begin_sym, setvar);
 setvar = add_set(if_sym, setvar);
 setvar = add_set(while_sym, setvar);
 return setvar ;
   }
 set add_block(setvar) /* called from procedure () ***/
 set setvar ;
 {
 setvar = add_set(semicolon_sym,setvar);
 setvar = add_set(period_sym, setvar);
 setvar = rm_set(ident_sym,setvar);
 return setvar;
  }
 set add_decls(setvar)
 set setvar;
 {
 setvar = add_set(colon_sym, setvar);
 setvar = add_set(integer_sym,setvar);
 setvar = add_set(real_sym, setvar);
 setvar = rm_set(semicolon_sym, setvar);
 return setvar;
   }

 set add_st_list(setvar) /* called from block() ***/
 set setvar;
 {

 setvar = add_set(end_sym, setvar);
 setvar = add_set(else_sym,setvar);
 setvar = add_set(semicolon_sym,setvar);
 setvar = add_set(ident_sym,setvar);
 return setvar;
   }

 set add_id_list(setvar) /*....from var_decl() ***/
 set setvar;
 {

 setvar = add_set(integer_sym, setvar);
 setvar = add_set(real_sym,setvar);
 setvar = add_set(comma_sym,setvar);
 setvar = add_set(colon_sym,setvar);
 return setvar;
  }
 set add_p_list(setvar) /* ... from proc_header() ***/

 set setvar;
```

```
{
setvar = add_set(rparen_sym, setvar);
setvar = add_set(semicolon_sym,setvar);
return setvar;
 }
set add_exp_list(setvar)    /* ... from statement()  ***/
set setvar;
{

setvar = add_set(rparen_sym, setvar);
setvar = add_set(comma_sym,setvar);
return setvar;
 }

set add_exp_assign(setvar) /*....from statement() ***/
set setvar;
{

setvar = add_set(semicolon_sym, setvar);
setvar = add_set(end_sym,setvar);
setvar = add_set(else_sym,setvar);
setvar = add_set(while_sym,setvar);
setvar = add_set(if_sym, setvar);
return setvar;
 }



set add_e_lst(setvar) /*....from var_decl() ***/
set setvar;
{

setvar = add_exp_assign(setvar);
setvar = add_set(comma_sym,setvar);
setvar = add_set(rparen_sym,setvar);
return setvar;
  }
set add_exp_while(setvar) /*....from statements() ***/
{

setvar = add_exp_assign( setvar);
setvar = add_set(do_sym,setvar);
return setvar;
 }
set add_exp_if(setvar) /*....from statements()   ***/

{

setvar = add_exp_assign(setvar);
setvar = add_set(then_sym,setvar);
return setvar;
 }

set add_s_exp(setvar) /*....from expressions() ***/
set setvar;
{
```

```
setvar = add_set(plus_sym, setvar);
setvar = add_set(minus_sym,setvar);
setvar = add_set(lt_sym,setvar);
setvar = add_set(lteq_sym,setvar);
setvar = add_set(eq_sym, setvar);
setvar = add_set(noteq_sym,setvar);
setvar = add_set(gteq_sym,setvar);
setvar = add_set(gt_sym,setvar);
setvar = add_set(rparen_sym, setvar);
setvar = add_set(comma_sym, setvar);
setvar = add_set(semicolon_sym,setvar);
setvar = add_set(end_sym,setvar);
setvar = add_set(else_sym,setvar);
setvar = add_set(do_sym, setvar);
setvar = add_set(then_sym, setvar);
return setvar;
 }

set add_term(setvar)
set setvar;

{

setvar = add_s_exp(setvar);
  /* call proc for simple_expression set */
setvar = add_set(mult_sym,setvar);
setvar = add_set(divide_sym,setvar);
setvar = add_set(div_sym,setvar);
return setvar;
 }
set add_factor(setvar)  /*....from statement() ***/
set setvar;
{

setvar = add_term(setvar);
return setvar;
 }
set add_exp_stat(setvar) /*....from statement()  */
set setvar;
{
setvar = add_set(rparen_sym, setvar);
setvar = add_set(semicolon_sym,setvar);
setvar = add_set(if_sym,setvar);
setvar = add_set(while_sym,setvar);
setvar = add_set(ident_sym, setvar);
return setvar;
 }
set add_not_proc(setvar) /*....from statement() */
set setvar;
{
setvar = rm_set(rparen_sym, setvar);
return setvar;
 }
```

```
                    /* Symbtab.c */
                     /* Symbol table generator */

 #include <stdio.h>
 #include <string.h>
#include "externs.h"

static debug =0;
void table_add()
{
++top;
symbol_table[top] = NULL ;
}
node_ptr add_ident(p, success,isvar, no_loc)
 node_ptr p;
 int  success, isvar ,no_loc ;
{
  list_ptr new_node;
  short cond;
if (debug) printf("in add_ident %d\n", no_loc);
if( p == NULL )
  { p= (sym_node *) malloc(sizeof(sym_node));
     if (symbol_table[top] == NULL )
         symbol_table[top] = p ;
     strcpy(p->ident_name, lex);
     p->left = p->right = NULL;
     p->ident_type = isvar ;
     if (isvar)
     {
     p->location = no_loc +2 ;
     new_node = (list_node *) malloc(sizeof(list_node));
     new_node->node = p ;
     new_node->next =  var_list ;
     var_list = new_node ;
}
 else
 p->no_param = 0;
 success = 1;
}

else
if ((cond = strcmp(lex, p->ident_name)) == 0)
{
 st_error(1);
success = 0 ;
}
else
 if (cond < 0)
   p->left = add_ident(p->left,success,isvar,no_loc) ;
   else
   p->right = add_ident(p->right,success,isvar, no_loc);
  return p;
}
 void generate_label() /* generate the lable for
                                       next procedure */
 {
  strcpy(proc_label, "$proc");
  proc_label[5] = proc_no++ ;
  printf ("label $s\n",proc_label);
 }

 void update_type(type, param)
 short type;
   short param;
{
list_ptr p;
for (p = var_list ; p!= NULL; p= p->next)
{
(p->node ) ->var_type = type ;
(p->node) ->param = param ;
}
}
void update_params(p,type)
 node_ptr p ;
short type;
 {
 short count = 0;
short i;
list_ptr temp =NULL ;

for (temp = var_list ; temp != NULL; ++count)
temp = temp ->next ;
 for (i = p->no_param; i<p->no_param + count; ++i)
 p->para_list[i] = type;
 if (debug) printf ("# %d count %d\n", p->no_param,count);
 p->no_param = p->no_param + count ;
 }
 node_ptr search(p)
 node_ptr p;

 {
 short cond ;
 if (p == NULL)
 return NULL;
else
if (( cond = strcmp(lex,p->ident_name)) == 0)

 return p ;

 else
if (cond < 0 )
 return search (p->left);
else
 return search(p->right);
}
 node_ptr table_search(t, level)
int t;
int  *level;
```

```
 { int i;
 node_ptr  pt= NULL;
 for (*level = i = t; i>=0; *level = --i)
  /*level is the level in the */
if (( pt = search(symbol_table[i])) !=NULL )
 /*symtab where var was found */
    return pt;
  st_error(2);
  return pt;
}
 void free_var_list()
{
 list_ptr p, q ;
 for (p = var_list; p !=NULL; p = q )
    {
 q = p->next ;
 free(p);
}
 var_list = NULL;
}
void free_table(p)
 node_ptr p;
{
 if (p != NULL)
{
if (p->left !=NULL)
{
 free_table(p->left);
 p->left = NULL ;
}
 if (p->right !=NULL)
{
 free_table(p->right);
  p->right = NULL;
}
if (p != NULL) free(p);
}
}
void print_table(p)
  node_ptr p;
{
 if ( p != NULL)
{
print_table(p->left);
 printf(" %s %d ", p->ident_name, p->location);
 print_table(p->right);
}
}
void install_procs()
{node_ptr proc = NULL;
int success;
strcpy(lex,"readi");
```

```
    add_ident(symbol_table[top],success,0);
    proc=search(symbol_table[top]);
    proc->ident_type=3;
    proc->no_param=1;
    proc->para_list[0]=0;
    free_var_list();

    strcpy(lex,"writei");
    add_ident (symbol_table[top],success,0);
    proc=search(symbol_table[top]);
    proc->ident_type=3;
    proc->no_param=1;
    proc->para_list[0]=0;
    free_var_list();

    strcpy(lex,"readr");
    add_ident(symbol_table[top],success,0);
    proc=search(symbol_table[top]);
    proc->ident_type=3;
    proc->no_param=1;
    proc->para_list[0]=1;
    free_var_list();

    strcpy(lex,"writer");
    add_ident(symbol_table[top],success,0);
    proc=search(symbol_table[top]);
    proc->ident_type=3;
    proc->no_param=1;
    proc->para_list[0]=1;
    free_var_list();

    strcpy(lex,"writetxt");
    add_ident(symbol_table[top],success,0);
    proc=search(symbol_table[top]);
    proc->ident_type=3;
    proc->no_param=1;
    proc->para_list[0]=3;
    free_var_list();
}
```

/* Typechk.c */

```
 #include "externs.h"

 void param_check(p,count, tp)
 node_ptr  p;
 int count;
 id_type tp ;

{
  if (( count + 1) <= p->no_param)
     if (p->para_list[count] == 0) /*if it's integer type...*/
      if (tp == int_type) ; /* do nothing -- it's fine.. */
      else
   st_error(10) ;
 else
  if (p->para_list[count] == 3) /* string */
    if (tp == str_type );
    else
    st_error(13);
else
 st_error(12);
 }
 id_type check_type(tp1, tp2)
   id_type tp1, tp2 ;
 {
 if (( tp1 != err_type) && (tp2 != err_type ))
   if (( tp1 == float_type) &&
 (tp2 == int_type || tp2 == float_type ))
  tp1 = float_type;
else
 if (( tp2 == float_type) && (tp2 == int_type ))
  tp1 = int_type;
else
if (( tp1 == int_type) && (tp2 == int_type))
   tp1 = int_type ;
else

 {
   st_error(5);
 tp1 = err_type ;
 }
else
{
  st_error (5) ;
 tp1 = err_type ;
 }
 return tp1;
}
 void assign_check(pt, tp2)
 node_ptr pt;
 id_type tp2;

 {
 id_type tp1;
 if (pt ->ident_type != 1)
  st_error(7);
 else
 {
  switch (pt->var_type)  {
 case 0 : tp1 = int_type ; break ;
 case 1 : tp1 = float_type ; break ;
 default : tp1 = err_type ;
 st_error(3);
}

 if ((tp1 != err_type) &&
 (tp2 == float_type || tp2 == int_type ))
  if ((tp1 == int_type) && ( tp2 == float_type))
  st_error(5);
}
}
```

/* Globals.h */

```
# define TRUE 1
# define MAXLEN 10
# define NO_KEYWORDS 13
# define MAX_STR_LEN 80

char lex[MAXLEN] ;
char strarry[MAX_STR_LEN];
typedef enum {
   lt_sym, eq_sym, gt_sym, lteq_sym, noteq_sym,gteq_sym,
   lparen_sym,rparen_sym, plus_sym,comma_sym,minus_sym,
   period_sym,assign_sym, colon_sym, semicolon_sym,
   begin_sym,do_sym,else_sym,end_sym, eof_sym, if_sym
   integer_sym,procedure_sym,real_sym,then_sym,
   var_sym,mult_sym,divide_sym,div_sym,while_sym, ident_sym,
   string_sym, error_sym,int_sym,float_sym,
   program_sym } token ;
 token sym ;
 token getsym();
 void error(), S_error(), ST_error(), print_tok();

 void table_add(), update_type(),free_var_list(),
  free_table(),print_table(),generate_lable();

 typedef struct sym_node *node_ptr ;
 typedef struct list_node *list_ptr ;
 typedef struct list_node
 {
 node_ptr node ;
 list_ptr next ;
  }
 list_node;
typedef struct sym_node
{
short ident_type ;
 char ident_name[10] ;
int label_no ;
short no_param ;
short para_list[10] ;
 short var_type ;
int location ;
 short param ;
node_ptr left ;
node_ptr right ;
 }
 sym_node ;
node_ptr add_ident(), search(), table_search();

 int int_val = 0 ;
float float_val = 0 ;
int line_no = 1 ;
node_ptr symbol_table[20] ;
list_ptr var_list = NULL ;
short top = -1 ;
```

```
typedef unsigned int set ;
int is_error = 0 ;
int proc_no = 0 ;
char proc_label[] = "$proc" ;
int while_count = 0 ;
char while_lab[] = "$while" ;
char W_end_lab[] = "$end";
int if_count = 0;
char if_end_lab[] = "$ifend" ;
char else_lab[] = "$else" ;
```

```
                    { sample run }
                    { sample.u }

program factorial;

 var  n: integer;
      result:integer;

 procedure fact (n, r: integer );
     var r1, n1: integer;
  begin
  if n<= 0then
  r:=1
              else
    n1:= n-1 ;
    fact (n1, r1);
    r := n * r1
    end
 end;

   begin
     writetxt (' Enter a number: ');
     readi(n);
    if n < 0 then
    writetxt ('Negative number entered\n' );

  else

   fact (n,result);
   writei(n)
   writetxt(' factorial is ' );
   writei(result);
   writetxt ('\n')
 end

end.
```

{ Output file sample.asm }

```
 $proc0
     alloc 2
     pushI 3
     fetchI
     pushI 0
     leI
     jumpz $else0
     pushI 4
     pushcI 1
     popiI
     jump $ifend0
              $else0
     pushI 3
     fetchI
     pushcI 1
     subI
     popI 6
     enter 2
     pusha 6
     pusha 5
     call $proc0, 2
     pushI 4
     pushI 3
     fetchI
     pushI 5
     mulI
     popiI
$ifend0
     return
     main
        enter 0
        alloc 2
        pushcI 16
        msg ' Enter a number :'
        inpI
        popI 3
        pushI 3
        pushcI 0
        itI
        jumpz $else1
        pushcI 24
        msg 'Negative number entered\n'
        jump $ifend1
$else1
        enter 1
        pusha 3
        pusha 4
        call $proc0, 2
        pushI 3
        outI
        pushcI 14
        msg 'factorial is '
        pushI 4
        outI
        pushcI 1
        msg'\n'
$ifend1
        return
```

BIBLIOGRAPHY

[1] Aho, Alfred V; Ullman, Jeffrey D — Principles of Compiler Design Addison-Weslet / Narosa

[2] Aho, Alfred V; Ullman, Jeffrey D — The Theory of Parsing, Translation and Compilation, Vol. I Parsing, Englewood Cliffs, N.J. Prentice-Hall

[3] Amman, U — On Code generation in PASCAL computer. Software Practise and Experience 7, 391-423 (1977)

[4] Bayer, F L; Eickel, J (Ed.) — Compiler Construction, An advanced Course, 2nd Ed., Springer-Verlag

[5] Bruno, J L; Lassange, T — The Generation of optimal code for stack machines, Journal of the ACM 22(j) 382-396 (1975)

[6] Bruno, J L; Sethi, R — Code Generation for a One Register Machine, Journal of the ACM 23(j), 382-396 (1976)

[7] Hopcroft, J E; Ullman, J D — Introduction to Automata theory, Language and computation, Addison-Wesley 1979

[8] Nori, K V; Amman, U; Jenson, K; Nagel, H H; Jacob, C — "Pascal-P implementation Notes" in Pascal - The language and implemeentation D.W Barron, ed. New York John Wiley & Sons

[9] Poole, P C — "Portable and Adaptable Compiler," in G Goos and J Harlmams, Lecture Notes in Computer Sciences, 2nd ed New York : Springer-Verlag 1976

[10] Stone, Chen, Flynn and others — Intoduction to computer architectures

[11] Tremblay, Jean Paul, Sorenson Paul, G	The theory and practice of Compiler writing McGraw Hill (Computer Sci. series)
[12] Waite, W M Goos, G	Compiler Construction, Texts and Monographs in computer science. Springer-Verlag