

College Engineering, Trivandrum.
Department of Computer Science and Engineering

[Question]

In this lab we are going to construct a compiler|interpreter for a simple imperative programming language. The CFG is given below.

Program-> PROG declarations BEGIN command_sequence END

declarations -> e |INTEGER id_seq IDENTIFIER.

id_seq \rightarrow e | id_seq IDENTIFIER,

command_sequence ->e | command_sequence command ;

command -> e

- | IDENTIFIER :=expression
- | IF exp THEN command_sequence ELSE command_sequence ENDIF
- | WHILE exp DO command_sequence ENDWHILE
- | READ IDENTIFIER
- | WRITE expression

expression -> NUMBER | IDENTIFIER | (expression)

- | expression + expression | expression * expression
- | expression - expression | expression / expression
- | expression = expression
- | expression < expression
- | expression > expression

where the non-terminal symbols are given in all lowercase and the terminal symbols are given in all caps or as literal symbols. The start symbol is *program*. There are two context sensitive requirements for the language, variables must be declared before they are referenced and a variable may be declared only once.

An example program is given below.

```
prog
  integer a,b.
begin
  read n;
  if a < 10 then b := 1; else; endif;
  while a < 10 do b := 5*a; a:= a+1; endwhile;
  write a;
  write b;
end
```

Assignment 1 (must be completed within 2 lab sessions(1st and 2nd lab sessions))

Generation of lexical analyzer and parser for our language.

Input: A program in our language.

Output: Whether a valid program or not .

Assignment 2 (must be completed within 2 lab sessions (3rd and 4th lab sessions))

Creating a symbol table for our language, (linked list implementation)

Input: A programming language.

Output: The contents of the symbol table.

Make modification to the parser module that we did in the first assignment by implementing the following functions whose prototype is given.

Structure of the symbol table is

```
struct sym_rec {
    char *name; //name of the symbol
    struct sym_rec next; //link field
    int data_offset; //will be used during code generation phase.
}*sym_record; //points to the first record
struct sym_rec * put_symbol(char * name); //puts an identifier into the table.
struct sym_rec * get_symbol(char * name); returns a pointer to the symbol table
entry
```

or a NULL pointer if not found.

void install(char *name); //installs a symbol into the symbol table if it is not in the symbol table using the above two functions. Reports appropriate error messages.

void context_check(char *name); //checks the context sensitive requirement of our language and if violated appropriate error messages.

Assignment 3

Code generation.

Input: A program in our Language.

Output: stack machine code written into a file.

The code is generated from the implicitly created parse tree. Here we are generating code for a virtual machine called a stack machine. The virtual machine consists of three segments. A data segment, a code segment and an expression stack. The data segment contains the values associated with the variables. Each variable is assigned to a location in the data segment which holds the associated value. The code segment consists of a sequence of operations, i.e. the stack machine code. Program constants are incorporated in the code segment since their values do not change. The expression stack is a stack which is used to hold intermediate values in the evaluation of expressions.

Instruction format for the stack machine

opcode	operand
--------	---------

Instruction set

(operand = 0 for opcodes with no operands. Eg. write, lt, gt, etc.)

Instruction	Operand	Meaning
res	n	Reserve n locations in the bottom of the stack(reserve space for data variables.)
read	n	stack(n) := input
write	0	output := stack(top--)
goto	n	Execute from the n th code onwards in the code segment
lt	0	If stack(top-1) < stack(top) stack(top) := 1 else stack(top):=0
gt	0	
eq	0	
jmp_false	n	If stack(top) = 0 execute the n th code in the code segment, top--
load_var	n	load variable at base offset n of the stack into top of the stack, top--
load_int	n	stack(++top) := n

Instruction	Operand	Meaning
store	n	store stack(top) at variable location n from base of the stack, top--
add	0	stack(top-1) := stack(top) + stack(top-1), top--
xor	0	
mul	0	
div	0	stack(top-1) := stack(top-1) / stack(top), top--
sub		
halt	0	stop execution

Translation Schemes

For declaration statements

Integer x,y,z. res 3

For statements

x := expression	code for expression store x
IF expression THEN	code for expression
command_sequence1	jmp_false L1
ELSE	code for command_sequence1
command_sequence2	
ENDIF	goto L2
	L1: code for command_sequence2
	L2:
WHILE expression DO	L1:code for expression
command_sequence	jmp_false L2
ENDWHILE	code for command_sequence
	goto L1
	L2:
read x	read x
x := expression	code for expression store x
Write expression	code for expression write

For expressions

constant	load_int constant
variable	load_var variable
expression1 op expression 2	code for expression1 code for expression 2 code for op

eg:

input:

```
prog
  Integer a,b.
begin
  Read a;
  if a < 10 then b := 1; else ; endif;
  while a < 10 do b :=5*a; a := a+1; endwhile;
  write a;
  write b;
end
```

output:

```
0:res      2
1:read     0
2:load_var 0
3:load_int 10
4:lt       0
5:jmp_false 9
6:ld_int   1
7:store    1
8:goto     9
9:load_var 0
10:load_int 10
11:lt      0
12:jmp_false 22
13:load_int 5
14:load_var 0
15:mul     0
16:store   1
17:load_var 0
18:load_int 1
19:add     0
20:store   0
21:goto    9
22:load_var 0
23:write   0
24:load_var 1
```

25:write	0
26:halt	0

Assignment 4

Design an interpreter for the stack machine.

Input: stack machine code.

Output: Execute the code and generate the results.

For our example code, output will be the final values of a and b.