**University of Central Florida**

**Department of Computer Science**

**COP 3402: System Software**

**Spring 2020**

**Homework #4 (PL/0 Compiler)**

**Due 2021 by 11:59 p.m.**

**This is a solo or team project (Same team as previous homeworks)**

**REQUIRMENT:**

**All assignments must compile and run on the Eustis server. Please see course website for details concerning use of Eustis.**

**Objective:**

In this assignment, you must extend the functionality of Assignment 3 to include the additional grammatical constructs highlighted in yellow in the grammar below.

**Example of a program written in PL/0:**

**var** x, w;

**begin**

x:= 4;

**read** w;

**if** w > x **then**

w:= w + 1

**else**

w:= x;

**write** w

**end.**

**Component Descriptions:**

The compiler must read a program written in PL/0 and generate code for the Virtual Machine (VM) you implemented in HW1. This assignment extends the functionality of the previous by including procedures and expanding on the if-then construct.

**Submission Instructions and rubric:**

1.- Submit via WebCourses:

1. Source code of the PL/0 compiler. You may have as many source code files as you desire, and with whatever names, but you must include a makefile. For one you can adapt to your implementation see HW4 resources file on webcourses.
2. A text file with instructions on how to use your program entitled readme.txt.
3. Only one submission per team: the name of all team members must be written in all source code header files, in a comment on the submission, and in the readme.
4. Include comments in your program
5. All files should be compressed into a single .zip format.
6. **Late assignments will not be accepted (for this project there is not a two day extension after the due date).**
7. Output should print to the screen and should follow the format in Appendix A. A deduction of 5 points will be applied to submissions that do not print to the screen.
8. The input file should be given as a command line argument. A deduction of 5 points will be applied to submissions that do not implement this.

**Please see the homework 3 instructions for output specifications. We will be again using a bash script for testing. Error handling and directives follow the same patterns as the last assignment.**

**Rubric**

15 – Compiles

20 – Produces some instructions before segfaulting or looping infinitely

5 – Follows IO specifications (takes command line argument for input file name and prints output to console)

5 – README.txt containing author names

5 – Supports directives

5 – Supports error handling

10 – Correctly implements return

10 – Correctly implements load and store with levels, supporting variables at different levels and variables with the same name

10 – Correctly implements call

10 – Correctly implements else

5 – Correctly implements convention of putting jumps to all procedures at the beginning of code

+5 – Follows formatting guidelines correctly, includes make file for testing

**Appendix A:**

**Traces of Execution:**

Example 1, if the input is:

procedure A;

var y;

begin

y := 12;

end;

begin

call A;

end.

The output should look like:

Lexeme Table:

lexeme token type

procedure 30

A 2

; 18

var 29

y 2

; 18

begin 21

y 2

:= 20

12 3

; 18

end 22

; 18

begin 21

call 27

A 2

; 18

end 22

. 19

Lexeme List:

30 2 A 18 29 2 y 18 21 2 y 20 3 12 18 22 18 21 27 2 A 18 22 19

Generated Assembly:

Line OP L M

0 JMP 0 7

1 JMP 0 2

2 INC 0 5

3 LIT 0 12

4 STO 0 4

5 LIT 0 0

6 RTN 0 0

7 INC 0 4

8 LIT 0 0

9 CAL 0 2

10 SYS 0 3

PC BP SP stack

Initial values: 0 0 -1

0 JMP 0 7 7 0 -1

7 INC 0 4 8 0 3 0 0 0 0

8 LIT 0 0 9 0 4 0 0 0 0 0

9 CAL 0 2 2 5 4 0 0 0 0 0

2 INC 0 5 3 5 9 0 0 0 0 0 | 0 0 10 0 0

3 LIT 0 12 4 5 10 0 0 0 0 0 | 0 0 10 0 0 12

4 STO 0 4 5 5 9 0 0 0 0 0 | 0 0 10 0 12

5 LIT 0 0 6 5 10 0 0 0 0 0 | 0 0 10 0 12 0

6 RTN 0 0 10 0 4 0 0 0 0 0

10 SYS 0 3 11 0 4 0 0 0 0 0

Example 2, see HW4 example output2.txt for example:

var x, y, z, v, w;

procedure a;

var x, y, u, v;

procedure b;

var y, z, v;

procedure c;

var y, z;

begin

z := 1;

x := y+z+w

end;

begin

y:=x+u+w;

call c

end;

begin

z:= 2;

u:=z+w;

call b

end;

procedure A;

var F, N;

procedure FACT;

var ANS1;

begin

ANS1 := N;

N := N - 1;

if N = 0 then F := 1;

if N > 0 then call FACT;

F := F \* ANS1;

end;

begin

N := 3;

call FACT;

write F;

end;

procedure poly (variable);

var total;

begin

return (variable \* variable + variable \* 2 + 9);

end;

begin

y :=2; z:=3;v:=4; w:=5;

x:= v+w;

write z;

call a;

call A;

v := 4 \* call poly (x \* z);

end.

**Appendix B:**

**EBNF of PL/0:**

program ::= block "**.**" **.**

block ::= const-declaration var-declaration procedure-declaration statement**.**

const-declaration ::= ["**const**" ident "**=**" number {"**,**" ident "**=**" number} "**;**"]**.**

var-declaration ::= [ "**var** "ident {"**,**" ident} “**;**"]**.**

procedure-declaration ::= { "**procedure**" ident [ “(“ ident “)” ] "**;**" block "**;**" }.

statement ::= [ ident "**:=**" expression

| "**call**" ident [ “(“ expression “)” ]

| “**return**” [ “(“ expression “)” ]

| "**begin**" statement { "**;**" statement } "**end**"

| "**if**" condition "**then**" statement ["**else**" statement]

| "**while**" condition "**do**" statement

| "**read**" ident

| "**write**" expression

| **e** ] **.**

condition ::= "**odd**" expression

| expression rel-op expression**.**

rel-op ::= "**=**"|“**<>**"|"**<**"|"**<=**"|"**>**"|"**>=**“**.**

expression ::= [ "**+**"|"**-**"] term { ("**+**"|"**-**") term}**.**

term ::= factor {("**\***"|"**/**"|”%”) factor}**.**

factor ::= ident | number | "**(**" expression "**)**“ | “**call**” ident [ “(” expression “)” ]**.**

number ::= digit {digit}**.**

ident ::= letter {letter | digit}**.**

digit ;;= "**0**" | "**1**" | "**2**" | "**3**" | "**4**" | "**5**" | "**6**" | "**7**" | "**8**" | "**9**“**.**

letter ::= "**a**" | "**b**" | … | "**y**" | "**z**" | "**A**" | "**B**" | ... | "**Y**" | "**Z**"**.**

**Based on Wirth’s definition for EBNF we have the following rule:**

**[ ] means an optional item.**

**{ } means repeat 0 or more times.**

**Terminal symbols are enclosed in quote marks.**

**A period is used to indicate the end of the definition of a syntactic class.**

**Appendix C:**

**Error messages for the tiny PL/0 Parser:**

* program must end with period
* const, var, procedure, call, and read keywords must be followed by identifier
* competing symbol declarations at the same level
* constants must be assigned with =
* constants must be assigned an integer value
* symbol declarations must be followed by a semicolon
* undeclared variable or constant in equation
* only variable values may be altered
* assignment statements must use :=
* begin must be followed by end
* if must be followed by then
* while must be followed by do
* condition must contain comparison operator
* right parenthesis must follow left parenthesis
* arithmetic equations must contain operands, parentheses, numbers, or symbols
* undeclared procedure for call
* parameters may only be specified by an identifier
* parameters must be declared
* cannot return from main

**These are all the error messages you should have in your parser.**

**Appendix D: Pseudocode**

GLOBAL VARIABLE procedurecount = 0

FINDPROCEDURE (index of the procedure i)

linear search through the symbol table looking at the value attribute of symbols with

kind = 3 (procedures), return the index of the value that matches

MARK (count)

start from the end of the symbol table, looping backwards,

if entry is unmarked, mark it & count--

else continue

SYMBOLTABLECHECK (name, level)

linear search through symbol table looking at name and level

return index if exact match for both is found unmarked, -1 if not

SYMBOLTABLESEARCH (name, lexlevel, kind)

linear search through symbol table looking at name and level

return index of exact match of name and kind, unmarked with nearest lexlevel

-1 if none found

PROGRAM

numProc = 1

emit JMP

foreach lexeme in list

if lexme.type = proceduresym

numProc++

emit JMP

add to symbol table (kind 3, "main", 0, 0, 0, unmarked, 0)

procedurecount++

BLOCK(0, 0, 0)

if token != .

error

for i = 0, i < numProc, i++

code[i].m = symboltable[FINDPROCEDURE(i)].addr

foreach line in code

if line.OP == 5 (CAL)

line.M = symboltable[FINDPROCEDURE(line number)].addr

emit halt

BLOCK (lexlevel, param, procedureIndex)

c = CONST-DECLARATION (lexlevel)

v = VAR-DECLARATION (lexlevel, param)

p = PROCEDURE-DECLARATION (lexlevel)

symboltable[procedureIndex].addr = current code index

emit INC (M = 4 + v)

STATEMENT(lexlevel)

MARK(c + v + p)

CONST-DECLARATION (lexlevel)

numConst = 0

if token == const

do

numConst++

get next token

if token != identsym

error

if SYMBOLTABLECHECK( token (the identifier), lexlevel) != -1

error

save ident name

get next token

if token != =

error

get next token

if token != number

error

add to symbol table (kind 1, saved name, number, lexlevel, 0, unmarked, 0)

get next token

while token == ,

if token != ;

error

get next token

return numConst

VAR-DECLARATION (lexlevel, param)

if param == 1

numVars = 1

else

numVars = 0

if token == var

do

numVars++

get nex token

if token != ident

error

if SYMBOLTABLECHECK (token, lexlevel) != -1

error

add to symboltable (kind 2, name, 0, lexlevel, var# + 3, unmarked, 0)

get next token

while token == ,

if token != ;

error

get next token

return numVars

PROCEDURE-DECLARATION (lexlevel)

numProc = 0

if token == procedure

do

numProc++

get next token

if token != ident

error

if SYMBOLTABLECHECK (token, lexlevel) != -1

error

procIdx = end of the symbol table

add to symbol table (kind 3, name, val = procedurecount, lexlevel, 0, unmarked, param 0)

procedurecount++

get next token

if token == (

get next token

if token != ident

error

add to symbol table (kind 2, name, val 0, lexlevel + 1, addr 3, unmarked, 0)

symboltable[procIdx].param = 1

get next token

if token != )

error

get next token

if token != ;

error

get next token

BLOCK(lexlevel + 1, 1, procIdx)

else

if token != ;

error

get next token

BLOCK (lexlevel + 1, 0, procIdx)

if code[current code index - 1].OP != 2 && code[current code index - 1].M != 0

emit LIT (M = 0)

emit RTN

if token != ;

error

get next token

while token == procedure

return numProc

STATEMENT (lexlevel)

if token == ident

symIdx = SYMBOLTABLESEARCH (name, lexlevel, kind 2)

if symIdx == -1

error

get next token

if token != :=

error

get next token

EXPRESSION(lexlevel)

emit STO (L = lexlevel - symboltable[symIdx].level, M = symboltable[symIdx].addr)

return

if token == call

get next token

if token != ident

error

symIdx = SYMBOLTABLESEARCH(name, lexlevel, kind 3)

if symIdx == -1

error

get next token

if token == (

get next token

if table[symIdx].param != 1

error

EXPRESSION (lexlevel)

if token != )

error

get next token

else

emit LIT 0

emit CAL (L = lexlevel - symboltable[symIdx].level, M = symboltable[symIdx].value)

return

if token == return

if lexlevel == 0

error

get next token

if token == (

get next token

EXPRESSION(lexlevel)

emit RTN

if token != )

error

get next token

else

emit LIT 0

emit RTN

return

if token == begin

do

get next token

STATEMENT (lexlevel)

while token == ;

if token != end

error

get next token

return

if token == if

get next token

CONDITION (lexlevel)

jpcIdx = current code index

emit JPC

if token != then

error

get next token

STATEMENT (lexlevel)

if token == else

get next token

jmpIdx = current code index

emit JMP

code[jpcIdx].M = current code index

STATEMENT (lexlevel)

code[jmpIdx].M = current code index

else

code[jpcIdx].M = current code index

return

if token == while

get next token

loopIdx = current code index

CONDITION (lexlevel)

if token != do

error

get next token

jpcIdx = current code index

emit JPC

STATEMENT (lexlevel)

emit JMP (M = loopIdx)

code[jpcIdx].M = current code index

return

if token == read

get next token

if token != ident

error

symIdx = SYMBOLTABLESEARCH (token, lexlevel, kind 2)

if symIdx == -1

error

get next token

emit READ

emit STO (L = lexlevel - symboltable[symIdx].level, M = symboltable[symIdx].addr)

return

if token == write

get next token

EXPRESSION (lexlevel)

emit WRITE

return

CONDITION (lexlevel)

if token == odd

get next token

EXPRESSION (lexlevel)

emit ODD

else

EXPRESSION (lexlevel)

if token == =

get next token

EXPRESSION (lexlevel)

emit EQL

else if token == <>

get next token

EXPRESSION (lexlevel)

emit NEQ

else if token == <

get next token

EXPRESSION (lexlevel)

emit LSS

else if token == <=

get next token

EXPRESSION (lexlevel)

emit LEQ

else if token == >

get next token

EXPRESSION (lexlevel)

emit GTR

else if token == >=

get next token

EXPRESSION (lexlevel)

emit GEQ

else

error

EXPRESSION (lexlevel)

if token == -

get next token

TERM (lexlevel)

emit NEG

while token == + || token == -

if token == +

get next token

TERM(lexlevel)

emit ADD

else

get next token

TERM (lexlevel)

emit SUB

else

if token == +

get next token

TERM (lexlevel)

while token == + || token == -

if token == +

get next token

TERM(lexlevel)

emit ADD

else

get next token

TERM (lexlevel)

emit SUB

TERM (lexlevel)

FACTOR(lexlevel)

while token == \* || token == / || token == %

if token == \*

get next token

FACTOR(lexlevel)

emit MUL

else if token == /

get next token

FACTOR(lexlevel)

emit DIV

else

get next token

FACTOR (lexlevel)

emit MOD

FACTOR(lexlevel)

if token == ident

symIdxV = SYMBOLTABLESEARCH(token, lexlevel, 2)

symIdxC = SYMBOLTABLESEARCH(token, lexlevel, 1)

if symIdxV == -1 && symIdxC == -1

error

else if symIdxC == -1 || (symIdxV != -1 && symboltable[symIdxV].level > symboltable[symIdxC].level)

emit LOD (L = lexlevel - symboltable[symIdxV].level, M = symboltable[symIdxV].addr)

else

emit LIT (M = symboltable[symIdxC].value)

else if token == number

emit LIT

get next token

else if token == (

get next token

EXPRESSION(lexlevel)

if token != )

error

get next token

else if token == call

STATEMENT(lexlevel)

else

error

**Appendix E:**

**Symbol Table**

Recommended data structure for the symbol.

typedef struct

{

int kind; // const = 1, var = 2, proc = 3

char name[10]; // name up to 11 chars

int val; // number

int level; // L level

int addr; // M address

int mark; // to indicate that code has been generated already for a block.

int param; // to indicate if the parameter for a procedure has been

declared

} symbol;

symbol\_table[MAX\_SYMBOL\_TABLE\_SIZE = 500];

For constants, you must store kind, name and value.

For variables, you must store kind, name, L and M.

For procedures, you must store kind, name, L and M.