University of Central Florida

**Department of Computer Science**

**COP 3402: System Software**

**Summer 2021**

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

**Due July 30th, 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 implement the intermediate code generator for PL/0. The **Intermediate Code Generator** is a program that takes, as input, the output from the Parser, i.e. the Symbol Table and parsed code. As output, it produces the assembly language for your Virtual Machine (HW1).

**Submission Instructions and rubric:**

Submit via WebCourses:

* Edited version of codegen.c with the function generate\_code() implemented. We will use the lex.o, compiler.h, driver.c and parse.o files provided to you during grading, so do not edit them. You may add supplementary functions and global variables as you wish, but the two global variables, instruction \*code and int codeidx must be implemented in order for the printcode function to work correctly.

Only one submission per team: the name of all team members must be written in all source code header files and in a comment on the submission.

**Late assignments will not be accepted (for this project there is not a two day extension after the due date).**

Output should print to the screen and should follow the format in the example test cases. A deduction of 5 points will be applied to submissions that do not print to the screen. Utilize printcode() to ensure correct formatting.

The generate\_code() function has two parameters it receives from driver when called: the symbol table and the lexeme list. The first entry of the symbol table is main and the end of the table is marked with a entry with kind -1. When it is passed to generate\_code(), all the entries of the symbol table are marked with value 1. The end of the lexeme list is marked by an entry with type -1. The lexeme list is first passed through a correctly implemented parse function so you can assume that there are no errors when it reaches generate\_code().

**Rubric**

20 – compiles

5 – LIT

5 – RTN

5 – NEG

5 – ADD, SUB

5 – MUL, DIV, MOD

5 – ODD

5 – EQL, NEQ, LSS, LEQ, GTR, GEQ

5 – LOD

5 – STO

5 – CAL

5 – INC

5 – JMP

5 – JPC

5 – write

5 – read

5 – halt

**Appendix A:**

**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 "**;**" block "**;**" }.

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

| "**call**" ident

| "**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 "**)**“.

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 B: Implementation Methodology**

The overall structure of your code generator is going to appear very similar to your parser. You should have a function for each non-terminal in the grammar and the flow of execution should move recursively through the procedures. Instead of recognizing errors, you will be “emit”-ing code (adding a line of code to the code array). There are twenty-four instructions that your program needs to emit. Here are some guidelines about where those emits will occur.

**LIT** – emited in two places: when literal number is used in an expression and when a constant is used in an expression.

**OPR**

* **RTN –** the last instruction of a procedure; the procedure calls block to generate it’s code, after this, but before it returns, return should be emitted; please note that main does not return, it halts.
* **NEG** – within expression, if an expression begins with a minus symbol, the negate instruction is emitted after the first term has been generated
* **ADD, SUB** – within expression, after both terms have been generated, the correct operator instruction is emitted
* **MUL, DIV, MOD** – within term, after both factors have been generated, the correct arithmetic instruction is emitted
* **ODD –** within condition, after the expression has been generated, the odd instruction is emitted
* **EQL, NEQ, LSS, LEQ, GTR, GEQ** – within condition, after both sides of a condition have been generated, the comparison instruction is emitted

**LOD** – load is emitted in factor when a variable is used in an expression; only vars are loaded, constants use LIT; it is important to determine which variable is being referenced as there can be multiple vars with the same name in the symbol table; a marked var is inaccessible; there may be multiple unmarked vars with the same name, but they will all have a unique lex level; the correct var in the symbol table has the desired name, is unmarked, and has the highest lex level; you must make sure to unmark symbols as they are declared and mark them when they become out of scope (when their procedure goes to return from block after statement is called), when generate\_code() is called, all the symbols are marked; M is taken directly from the symbol table entry, L is the difference between the current lex level and the entry’s lex level

**STO –** store is emitted in two places: after a read instruction and after the expression in an assignment statement has been generated; again only vars can be stored and it’s important to determine which variable is being referenced, the guidelines for selection are the same as with a LOD and the M and L values for stores are determined in the same manner as loads

**CAL** – call is emitted in a call statement; only procedures can be called and once again it’s important to correctly determine which procedure is being called, the guidelines are the same as when picking a var to load; the L value is determined in the same manner as with loads and stores, but the M value is different; the M value for a call is the line number of the first instruction in the procedure, there are a number of methods for keeping track of these values; TA Elle recommends that you use the value field of the symbol table to note the line number of the first instruction in the procedure; the first instruction will always be an INC which is emitted in block before statement is called, so before you emit the call, store the value of the current code index in the value field of the current procedure; this way, when you go to call the procedure you can get the M value from the symbol table; we guarantee that a procedure will never be called before it’s code has begun emitting (a procedure can still call itself, but for example if there are two subprocedures of main, the first can’t call the second).

**INC** – the first instruction of all procedures, emitted in block before statement is called, M is equal to 3 plus the number of variables in the procedure

**JMP** – emitted in three places: at the very beginning, after the statement has been generated in while, and in if statements if an else is present, before the else statement has been generated; in the while case, you need to save the current code index before calling condition as this is the M value for the JMP; in the if-else case, if there is an else, emit the JMP before fixing the JPC and before generating code for the statement, you cannot know what the M value will be when the JMP is emitted in this case so you must save the current code condition and fix it later after the call to statement has returned; the JMP at the very beginning is to the first instruction of main, the M value cannot be known at the time of generation, but using the same method as in CAL for determining M value, we can fix the JMP at the very end using the value field of main in the symbol table

**JPC –** emitted in two places: after the condition has been generated in if statements and while statements; it’s important to note that in both cases the M value **cannot** be determined when the instruction is emitted so you must note the code index before emitting the JPC then later (in the while case, after the JMP following the statement has been generated; in the if case, after the statement before else has been generated) you can go back and set the M value for the JPC in the code array

**SYS 1 (write) –** emitted in write statements after the expression has been generated

**SYS 2 (read) –** emitted in read statements before the store instruction

**SYS 3 (halt) –** emitted at the very end of the program when the very first call to block returns to program.

Pseudocode for if statements:

if token == ifsym

get next token

CONDITION()

jpcIdx = codeIdx

emit JPC

get next token

STATEMENT()

if token == elsesym

get next token

jmpIdx = codeIdx

emit JMP

code[jpcIdx].M = codeIdx

STATEMENT()

code[jmpIdx].M = codeIdx

else

code[jpcIdx].M = codeIdx

Pseudocode for while statements:

if token == whilesym

get next token

jmpIdx = codeIdx

CONDITION()

get next token

jpcIdx = codeIdx

emit JPC

STATEMENT()

emit JMP // M = jmpIdx

code[jpcIdx].M = codeIdx