University of Central Florida

**Department of Computer Science**

**COP 3402: System Software**

**Summer 2022**

**Homework #3 Parser- Code Generator**

**This is a solo or team project (Same team as HW1 and HW2)**

**Due July 17th, by 11:59 p.m.**

**REQUIRMENT:**

**All assignments must compile and run on the Eustis3 server. Please see course website for details concerning use of Eustis3.**

**Objective:**

In this assignment, you must implement a Recursive Descent Parser and Intermediate Code Generator for PL/0. The **Parser** is a program that reads in the output of the Scanner (HW2) and parses the tokens. It must be capable of reading in the tokens produced by your Scanner (HW2) and produce, as output, if the program does not follow the grammar, a message indicating the type of error present **(the parser will never be given scanner output that contains errors; if an error is found in the scanner output, the driver will not call the parser)**. A list of the errors that must be considered can be found in Appendix C. In addition, the Parser must fill out the Symbol Table, which contains all of the variable, procedure and constant names within the input program. See Appendix E for more information regarding the Symbol Table. If the program is syntactically correct and the Symbol Table is created without error, the execution of the compiler continues with intermediate code generation. The **Intermediate Code Generator** uses the Symbol Table and Token List to translate the program into instructions for the VM. As output, it produces the assembly language for your Virtual Machine (HW1). Once the code has been generated for your Virtual Machine, the driver will call the virtual machine to execute the code. You may implement these two programs separately or combined. The pseudocode we provide has them interleaved.

**The Compile Driver:**

The **compile driver** is a program that manages the parts of the compiler. It handles the input, output, and execution of each of the parts of the project. First, it reads in the input file and stores the input program as a string. Next, it calls the scanner (HW2) and passes the input string. The scanner will return the token list, unless it found an error in which case it will return null. If null was returned by the scanner, the driver will end execution. Otherwise, the driver will call the parser/code generator (HW3) and pass the token list. Your parser/code generator should return the assembly instructions, unless an error was detected in which case it should return null. Again, if null is returned by the parser/code generator, the compile driver will end execution. Otherwise, it will call the virtual machine (HW3) and pass the code. The virtual machine will not return anything, but once it finishes, the driver will end execution by freeing any remaining dynamic memory.

The compiler driver has been provided for you. Additionally, compiled versions of the scanner and virtual machine are provided. It is our desire for you to focus on implementing this homework, rather than revising the previous homeworks. Unless you provide a scanner and virtual machine in your submission and indicate in your readme that we should use the files you provide, we will use the compiled versions in our grading.

**compiler.h:**

In addition to the driver.c file, we provide you with compiler.h. This is the file that allows the project to have multiple c files that can interact. Each c file in the project must have the line “#include compiler.h” at the top. You shouldn’t need to alter compiler.h, but you may if so desired. compiler.h should contain the function declarations for all functions you want to be accessible across files as well as any structs. Note that it includes the enum token\_type that you used in HW2; this means you can use the identifiers for the token\_types instead of their numerical values in your code (ie “elsesym” rather than “1”).

**Error Handling**

When your program encounters an error, it should print out an error message and stop parsing immediately. This is different than how your scanner handled errors. Do not continue parsing after an error is found.

**Submission Instructions:**

Submit via WebCourses:

1. Your source code. Because we’ve outlined an approach using one c file, we assume you will only submit parser.c, but you may have as many source code files as you desire. It is essential that you leave a note in your readme and as a comment on your submission if you submit more c files or you alter the provided files; you may lose points if you don’t.
2. A text file with instructions on how to use your program entitled readme.txt. If you didn’t change the driver or submit extra files, your readme can simply be your name (you still need to submit one).
3. Late policy is the same as HW1 and HW2: 10 points for one day, 20 for two
4. 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.

Appendix A

***Input Program:***

const a := 3;

var x;

procedure B;

var a;

begin

a := 10;

x := a - x;

end;

begin

x := -(((a\*8)+1)/5);

call B;

end.

***Command:*** ./a.out eft.txt

***Output:***

Lexeme Table:

lexeme token type

const 28

a 2

:= 20

3 3

; 18

var 29

x 2

; 18

procedure 30

B 2

; 18

var 29

a 2

; 18

begin 21

a 2

:= 20

10 3

; 18

x 2

:= 20

a 2

- 5

x 2

; 18

end 22

; 18

begin 21

x 2

:= 20

- 5

( 15

( 15

( 15

a 2

\* 6

8 3

) 16

+ 4

1 3

) 16

/ 7

5 3

) 16

; 18

call 27

B 2

; 18

end 22

. 19

Token List:

28 2 a 20 3 3 18 29 2 x 18 30 2 B 18 29 2 a 18 21 2 a 20 3 10 18 2 x 20 2 a 5 2 x 18 22 18 21 2 x 20 5 15 15 15 2 a 6 3 8 16 4 3 1 16 7 3 5 16 18 27 2 B 18 22 19

Symbol Table:

Kind | Name | Value | Level | Address | Mark

---------------------------------------------------

3 | main | 0 | 0 | 9 | 1

1 | a | 3 | 0 | 0 | 1

2 | x | 0 | 0 | 3 | 1

3 | B | 0 | 0 | 1 | 1

2 | a | 0 | 1 | 3 | 1

Assembly Code:

Line OP Code OP Name L M

0 7 JMP 0 27

1 6 INC 0 4

2 1 LIT 0 10

3 4 STO 0 3

4 3 LOD 0 3

5 3 LOD 1 3

6 2 SUB 0 3

7 4 STO 1 3

8 2 RTN 0 0

9 6 INC 0 4

10 1 LIT 0 3

11 1 LIT 0 8

12 2 MUL 0 4

13 1 LIT 0 1

14 2 ADD 0 2

15 1 LIT 0 5

16 2 DIV 0 5

17 2 NEG 0 1

18 4 STO 0 3

19 5 CAL 0 3

20 9 SYS 0 3

VM Execution:

PC BP SP stack

Initial values: 0 63 62

JMP 0 27 27 63 62

INC 0 4 30 63 66 0 0 0 0

LIT 0 3 33 63 67 0 0 0 0 3

LIT 0 8 36 63 68 0 0 0 0 3 8

MUL 0 4 39 63 67 0 0 0 0 24

LIT 0 1 42 63 68 0 0 0 0 24 1

ADD 0 2 45 63 67 0 0 0 0 25

LIT 0 5 48 63 68 0 0 0 0 25 5

DIV 0 5 51 63 67 0 0 0 0 5

NEG 0 1 54 63 67 0 0 0 0 -5

STO 0 3 57 63 66 0 0 0 -5

CAL 0 3 3 67 66 0 0 0 -5

INC 0 4 6 67 70 0 0 0 -5 | 63 63 60 0

LIT 0 10 9 67 71 0 0 0 -5 | 63 63 60 0 10

STO 0 3 12 67 70 0 0 0 -5 | 63 63 60 10

LOD 0 3 15 67 71 0 0 0 -5 | 63 63 60 10 10

LOD 1 3 18 67 72 0 0 0 -5 | 63 63 60 10 10 -5

SUB 0 3 21 67 71 0 0 0 -5 | 63 63 60 10 15

STO 1 3 24 67 70 0 0 0 15 | 63 63 60 10

RTN 0 0 60 63 66 0 0 0 15

SYS 0 3 63 63 66 0 0 0 15

**Appendix B: The Grammar**

**EBNF of tiny 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

| **ε** ] **.**

condition ::= 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.**

**This grammar is the ULTIMATE authority. It’s possible that lex.o or vm.o or the pseudocode or the examples have errors, but this does not. It is the basis of the whole project. There is an interesting quirk with the semicolon on the last statement in a begin-end: it’s optional. It can be present and it can be absent, but neither case should cause an error. This is because statement can be empty. Don’t stress too much about this if you don’t understand, it’s not a separate thing you have to account for, it’s innate to the grammar.**

**Appendix C: Error Messages**

**There are three types of error messages in PL/0:**

1. Errors generated based on the absence of an expected symbol: you check for a symbol and if it’s not present, you issue the error; the first 12 errors below are this type
2. Errors generated based on the presence of an unexpected symbol: you check for a symbol and if it’s not present, you look at the symbol that’s there instead and select the error based on what that symbol is; errors 13, 14, 15, 16, and 17 are this type
3. Errors generated due to conflicts with the symbol table: when you encounter an identifier you must check the symbol table to see if it can be used in that location; the last two errors are this type

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

1. Program must be closed by a period – found when the flow of control returns to **program** and the current symbol is not a period
2. Constant declarations should follow the pattern **ident ":=" number {"," ident ":=" number} –** found when the flow of control is in **const-declaration** and ident, :=, or number are missing
3. Variable declarations should follow the pattern **ident {"," ident}** – found when the flow of control is in **var-declaration** and ident is missing
4. Procedure declarations should follow the pattern **ident ";"** – found when the flow of control is in **procedure-declaration** and ident or ; is missing before **block** is entered
5. Variables must be assigned using := - found in **statement** in the assignment case when := is missing
6. Only variables may be assigned to or read – found in **statement** in the read case when the identifier is missing OR the identifier present is not a variable (does not have kind 2) and in the assignment case when the identifier is not a variable
7. call must be followed by a procedure identifier – found in **statement** in the call case when the identifier is missing OR the identifier present is not a procedure (does not have kind 3)
8. if must be followed by then – found in **statement** in the if case when flow of control returns from **condition** and the current symbol is not then
9. while must be followed by do – found in **statement** in the while case when flow of control returns from **condition** and the current symbol is not do
10. Relational operator missing from condition – found in **condition** in the case when odd was not found and flow of control returned from **expression** without error and the current symbol is not a relational operator
11. Arithmetic expressions may only contain arithmetic operators, numbers, parentheses, constants, and variables – found in **factor** when the current symbol is neither a number, an identifier, nor a ( OR when an identifier is found, but it is a procedure (kind 3)
12. ( must be followed by ) – found in **factor** in the parenthesis case when flow of control returns from **expression** without error, but a ) is not found
13. Multiple symbols in variable and constant declarations must be separated by commas – found in **var-declaration** and **const-declaration** when you check for the ending semicolon and find an identifier instead
14. Symbol declarations should close with a semicolon – found in **var-declaration** and **const-declaration** when you check for the ending semicolon and don’t find it OR an identifier; also found in **procedure-declaration** after flow of control returns from **block** and the semicolon is not present
15. Statements within begin-end must be separated by a semicolon – found in **statement** when the end symbol is expected but one of the following is found instead: identifier, read, write, begin, call, if, or while
16. begin must be followed by end – found in **statement** when the end symbol is expected and the symbol present is neither end, identifier, read, write, begin, call, if, nor while
17. Bad arithmetic – found at the end of **expression** before flow of control is returned to the caller when the current symbol is one of the following: ( identifier number Unlike the other errors of type B, there is not necessarily an error to be found in this location, so there is no alternative to this error
18. Conflicting symbol declarations – found in one of the declarations when the identifier being declared is already present and unmarked in the symbol table at the same lexical level
19. Undeclared identifier – found in **statement** (in the assignment, read, and call cases) or in **factor** (in the identifier case) when the identifier cannot be found in the symbol table unmarked

**Please note that we will check for the correct implementation of all of these errors. There is a function in parser.c which will print the error message for you and free the code array and symbol table. DO NOT ALTER THE ERROR LIST.**

**All errors should checked for at least once, some may have checks in multiple locations.**

**Appendix D: Pseudocode (parsing and code generation combined)**

This pseudocode is incomplete. It covers the most complex issues like when you should emit each instruction and the calls between functions, but it doesn’t specify where errors are or the movement of the current token pointer. Some issues are explained without pseudocode. You can extrapolate the token movement from the EBNF Grammar, each syntactic class is a function below. The error list specifies where each error should be recognized.You can use the labels from the token\_type enum. See end for FAQs

PROGRAM

emit JMP (M = 0, because we don’t know where the code for main will start)

add to symbol table (kind 3, “main”, 0, level = 0, 0, unmarked)

level = -1

BLOCK

emit HALT

now we know where our procedures start (because they’ve been processed and we

saved their starting code addresses in the symbol table), so we can fix the M

values for the CAL instructions and the M value of the initial JMP to main

make sure to multiply the new M values by 3 because they will be moved to

the PAS

BLOCK

Increment level

at this point we know where the current procedure is in the symbol table, it was the

last symbol added (current table index - 1), we want to save this before we

process more symbols

CONST-DECLARATION

x = VAR-DECLARATION (x is how many variable spaces we’ll need on the stack)

PROCEDURE-DECLARATION

we’re about to start emitting code for the current procedure, this is the point where

CAL instructions for this procedure will jump to, so we need to save this in

the symbol table in the addr field for this procedure

emit INC (M = x + 3)

STATEMENT

MARK

Decrement level

CONST-DECLARATION

VAR-DECLARATION

should return the number of variables declared so we can INC the right amount

when variables are added to the symbol table, we want to be able to use the addr

field for M in LOD and STO instructions, to achieve this, the addr field

should equal the number of variables before the one being added + 3

so the first variable in a procedure will have addr = 0 + 3, the second will

be addr = 1 + 3, and so on

PROCEDURE-DECLARATION

For each procedure:

1 - process the declaration (“procedure ident ;”)

2 - add it to the symbol table

3 - call BLOCK

4 - process the second semicolon

5 - emit RTN

STATEMENT

assignment statements (“ident := expression”)

make sure the identifier matches a variable available in the symbol table

EXPRESSION

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

call statements (“call ident”)

make sure the identifier matches a procedure available in the symbol table

emit CAL (L = level - table[symIdx].level, M = symIdx) (make sure the M

value is the index of the procedure in the symbol table so when we

correct the CAL instructions at the very end, we know which

procedures are being CAL’d)

begin statements (“begin statement { ; statement } end”)

do

process semicolon

STATEMENT

while token == semicolonsym

if statements (“if condition then statement [else statement]”)

CONDITION

jpcIdx = current code index

emit JPC (M = 0, because we don’t know where in the code we’ll jump to;

we don’t know how long the true actions are and we don’t know if

there are false actions)

STATEMENT

if token == elsesym

jmpIdx = current code index

emit JMP(M =0 because we don’t know how long the else section is)

code[jpcIdx].m = current code index \* 3 (because the PAS in the vm)

STATEMENT

code[jmpIdx].m = current code index \* 3 (because the PAS)

else

code[jpcIdx].m = current code index \* 3 (because the PAS)

while statements (“while condition do statement”)

loopIdx = current code index (because we’ll check our condition every time

we loop and this is where the condition code will start)

CONDITION

jpcIdx = current code index

emit JPC (M = 0, because this jumps to the end of the loop in the case that

the condition is false, but we don’t know how long the loop code is)

STATEMENT

emit JMP M = loopIdx \* 3 (because the PAS in the vm)

code[jpcIdx].m = current code index (because the PAS in the vm)

read statements (“read ident”)

make sure the identifier matches a variable available in the symbol table

emit READ

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

write statements (“write expression”)

EXPRESSION

emit WRITE

CONDITION

EXPRESSION

if-else-if structure with a branch for each rel-op ala:

else if token == geqsym

EXPRESSION

emit GEQ

EXPRESSION

if token == minussym

TERM

emit NEG

else

TERM

while token == plussym || token == minussym

if token == plussym

TERM

emit ADD

else

TERM

emit SUB

TERM

FACTOR

while token == multsym || token == slashsym || token == modsym

if token == multsym

FACTOR

emit MUL

else if token == slashsym

FACTOR

emit DIV

else

FACTOR

emit MOD

FACTOR

if token == identsym

try to find the identifier in the symbol table, it must be unmarked and it

cannot be a procedure, but both variables and constants are valid

options. It is possible that there are multiple entries with the same

identifier, however, they will always have different levels. Prioritize

the option (variable or constant) with the highest level value

if the identifier corresponds to a constant

emit LIT M = table[symIdx].val

else (meaning it corresponds to a variable)

emit LOD (L = level - table[symIdx].level, M = table[symIdx].addr)

else if token == numbersym

emit LIT (M = token.value)

else if token == lparentsym

EXPRESSION

**FAQs**

* How do you know what lexical level you’re at?
  + This can be a global variable or it can be passed or maybe you can come up with another way we haven’t thought of. It should start at -1 and then increment when you enter BLOCK and decrement when you return from BLOCK
* How should errors be handled?
  + Make sure you call the error printing function with the correct error code, it will free the symbol table and code array. Then you should stop executing. We don’t really care how you handle the stopping of execution, but we prefer that you avoid using system calls.
* What does emit mean?
  + It’s a simple “add an instruction to the code array and increment the code index”, it can actually be found in the slide decks. The instruction values are passed as arguments
* Some of the functions don’t have values specified for some fields, what’s up with that?
  + Sometimes it’s assumed by the nature of the instruction (like HALT is 9 0 3 all the time). Other times, it’s because it doesn’t matter. Like the very first JMP instruction doesn’t have an M value specified, it’s because it’s jumping to the first instruction of main and we can’t possibly know that when we emit it, but we need to reserve that space. At the end of PROGRAM it’s corrected.
* How does MULTIPLEDECLARATIONCHECK work?
  + This is a function given in the skeleton which is useful in CONST-DECLARATION, VAR-DECLARATION, and PROCEDURE-DECLARATION to check if an identifier has already been used
  + This function does a linear pass through the symbol table looking for the identifier given. If it finds that name, it checks to see if it’s unmarked (no? keep searching). If it finds an unmarked instance, it checks the level. If the level is equal to the current level, it returns that index. Otherwise it keeps searching until it gets to the end of the table, and if nothing is found, returns -1
  + A return value of -1 means that the identifier can be used, otherwise you’ve found an error (error 18 specifically)
* How does FINDSYMBOL work?
  + This is another function given in the skeleton which you can use in STATEMENT and FACTOR to find a symbol table entry. It returns -1 if the desired identifier can’t be found, the index of the identifier in the symbol table otherwise
  + This function does a linear search for the given name. An entry only matches if it has the correct name AND kind value AND is unmarked. Then it tries to maximize the level value
* How does MARK work?
  + This is another function given in the skeleton which you can use in BLOCK to mark the procedure’s sub symbols after you’ve finished processing the procedure
  + This function starts at the end of the table and works backward. It ignores marked entries. It looks at an entry’s level and if it is equal to the current level it marks that entry. It stops when it finds an unmarked entry whose level is less than the current level

**Appendix E:**

**Symbol Table**

Recommended data structure for the symbol table.

typedef struct

{

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

char name[12]; // name up to 11 chars

int value; // number (for constants)

int level; // L level

int addr; // M address

int mark;

} symbol;

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

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

For variables, you must store kind, name, level, addr, and mark.

For procedures, you must store kind, name, level, addr, and mark.

Unmarked and marked are arbitrary values; it doesn’t really matter as long as you’re consistent. We recommend 1 and 0.