University of Central Florida
Department of Computer Science
COP 3402: System Software
Fall 2021

Homework #3 (Parser- Code Generator)

Due October 31, 2021 by 11:59 p.m.

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

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. In addition, you must create a compiler driver to combine all the compiler parts into one single program.

Component Descriptions:
The **compiler driver** is a program that manages the parts of the compiler. It handles the input, output, and execution of the Scanner (HW2), the Parser (HW3), the Intermediate Code Generator (HW3) and the Virtual Machine (HW1). The compiler driver has been provided for you. Additionally, compiled implementations of HW1 and HW2 have been provided, so you can focus on the programs for this assignment, but you will have to correct them for HW4.

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 and it must be printed **(reminder: if the scanner detects an error the compilation process must stop and the error must be indicated**, **the compilation process must stop)**. 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 variables, procedure and constants names within the PL/0 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. (See Appendix D for parser pseudocode)

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 execution of the compiler driver continues by executing the generated assembly code on your Virtual Machine

The compiler driver supports the following compiler directives:

-l : print the list and table of lexemes/tokens (HW2 output) to the screen

-s : print the symbol table

-a : print the generated assembly code (parser/codegen output) to the screen

-v : print virtual machine execution trace (HW1 output) to the screen

<filename>.txt : input file name, for e.g. input.txt

Example commands:

./a.out input.txt –l –a –v	Print all types of output to the console
./a.out input.txt –v	Print only the VM execution trace to the console
./a.out input.txt	Print nothing to the console except for program read and write instructions.

<u>Notes on Implementation</u>

Our policy in grading is this: if it works and you didn't cheat, we don't care how you did it. If you choose to alter the provided files or submit your own lex.c vm.c instead of using the .o files or even include additional c files, you can! Just make sure to leave an explanation in your readme and the comment on your submission. There are many ways to implement this assignment. In the pseudocode in Appendix D, we've taken the interleaved approach (combining parser and code gen), but you can implement them separately if desired.

<u>Error Handling</u>

When your program encounters an error, it should print out an error message and stop executing immediately.

We will be using a bash script to test your programs. We've included printing functions for you to use; if you choose to alter them, you won't lose points as long as you output the necessary information, but you will delay the grading process. We use diff -w -B for evaluation.

Submission Instructions:

Submit via WebCourses:

1. Source code of the tiny- PL/0 compiler. Because we've outlined an approach using one 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.
3. Please don't compress your files
4. Late policy is the same as HW1 and HW2: 10 points for one day, 20 for two
5. 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.
6. Include comments in your program

What we're giving you:

- parser.c – this is a skeleton with the print functions implemented and some global variables; to print an error message pass the error number from Appendix C, you should always print error messages if they occur, after printing an error, the function will free the code array and symbol table for you, and you should return null to the driver; make sure that you only call the print code function or print symbol table functions IF their respective flags are true **There is a line commented out in parser which marks the end of the code array. YOU MUST UNCOMMENT IT IN ORDER FOR vm.o TO FUNCTION. IT WILL SEGFAULT IF YOU DON'T**
- driver.c – we've left this uncompiled in case you want to understand how it works, but you shouldn't need to alter it. It reads the input into a string from a file whose name is given as a command line argument. It reads in the compiler directives which are given as command line arguments. Then it calls the lexanalyzer function by passing the input string and a flag variable indicating whether output should be printed. If there were errors, it stops; otherwise, it calls the parse function by passing the lexeme list returned by the lexanalyzer and some flag variables. If there were errors, it stops; otherwise, it calls the vm function by passing the code returned by the parse and some flag variables. Then it frees everything and stops

You shouldn't have to write ANY free calls; they have all been implemented for you.

- lex.o and vm.o – these are compiled implementations of HW2 and HW1 respectively. They are correct. If you discover any bugs, please email TA Elle. They were compiled on Eustis3, so they will only run properly on Eustis3.
- Makefile – this will compile the program. It runs the command "gcc parser.c driver.c lex.o vm.o -lm", you can run the makefile with command "make"
- compiler.h – this is the most important bit. It allows the separate files to call each other's functions. You shouldn't need to alter it at all. If you do, leave a note.

- tester.sh – this is a sample bash script, similar to the one we use in grading. You can run it with the command “bash tester.sh” It tries to compile your program and stops if it doesn’t. Then it runs each test case, dumps the output to a file and then compares that file to the correct output using diff -w -B before printing out the results. If you’re not passing because of a formatting difference, you won’t lose points, but it will cause a delay in grading.
- Some test cases. They don’t cover everything, and we will use completely different ones during grading, so you should make up your own test cases to try.

Rubric

15 – Compiles
20 – Produces some instructions before segfaulting or looping infinitely, not necessarily correct, but enough to demonstrate that your program is doing something.
05 – README.txt containing author names and a description of alterations to the provided documents if present
20 – all errors are implemented correctly
10 – correctly parses symbol table
05 – correctly implements while structure
05 – correctly implements if structure
05 – correctly implements read and write instructions
05 – correctly implements arithmetic expressions
10 – correctly implements procedures (load, store, call, rtn, inc)

Appendix A: Examples

With the skeleton, we've included four sample tests. Here is a description of each:

- **basic.txt**
  - **output file – bout.txt**
  - **directives – a, s**
  - **what does it do? – This is a very simple test case**
- **errorA.txt**
  - **output file – outA.txt**
  - **what does it do? – This case is an example of error 11**
- **errorB.txt**
  - **output file – outB.txt**
  - **what does it do? – This case is an example of error 16**
- **errorC.txt**
  - **output file – outC.txt**
  - **what does it do? – This case is an example of error 19**
- **tip.txt**
  - **output file – tipout.txt**
  - **input numbers – '1 10 51 17 2 10 51 17 0'**
  - **directives – a, s**
  - **what does it do? – This is a really complex test case which asks for the dollar and change amounts and a tip percentage and then prints out either the tip or the total depending on user specification. This test doesn't cover everything, but it does cover a lot.**

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 ::= "**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.

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 to 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:

A. 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
B. 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
C. 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

<u>Error messages for the tiny PL/0 Parser:</u>

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 odd 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)

Note the use of labels from the token_type enum. See end for FAQs

```
PROGRAM
	emit JMP
	add to symbol table (kind 3, "main", 0, level = 0, 0, unmarked)
	level = -1
	BLOCK
	if token != periodsym
		error
	emit HALT
	for each line in code
		if line has OPR 5 (CALL)
			code[line].m = table[code[line].m].addr
	code[0].m = table[0].addr

BLOCK
	Increment level
	procedure_idx = current symbol table index - 1
	CONST-DECLARATION
	x = VAR-DECLARATION
	PROCEDURE-DECLARATION
	table[procedure_idx].addr = current code index * 3
	if level == 0
		emit INC (M = x)
	else
		emit INC (M = x + 3)
	STATEMENT
	MARK
	Decrement level

CONST-DECLARATION
	if token == const
		do
			get next token
			if token != identsym
				error
			symidx = MULTIPLEDECLARATIONCHECK(token)
			if symidx != -1
				error
			save ident name
			get next token
			if token != assignsym
				error
			get next token
```

```
                    if token != numbersym
                        error
                    add to symbol table (kind 1, saved name, number, level, 0, unmarked)
                    get next token
                while token == commasym
                if token != semicolonsym
                    if token == identsym
                        error
                    else
                        error
                get next token
```

VAR-DECLARATION

```
        numVars = 0
        if token == varsym
            do
                numVars++
                get next token
                if token != identsym
                    error
                symidx = MULTIPLEDECLARATIONCHECK(token)
                if symidx != -1
                    error
                if level == 0
                    add to symbol table (kind 2, ident, 0, level, numVars-1, unmarked)
                else
                    add to symbol table (kind 2, ident, 0, level, numVars+2, unmarked)
                get next token
            while token == commasym
            if token != semicolonsym
                if token == identsym
                    error
                else
                    error
            get next token
        return numVars
```

PROCEDURE-DECLARATION

```
        while token == procsym
            get next token
            if token != identsym
                error
            symidx = MULTIPLEDECLARATIONCHECK(token)
            if symidx != -1
                error
            add to symbol table (kind 3, ident, 0, level, 0, unmarked)
            get next token
            if token != semicolonsym
```

```
                error
            get next token
            BLOCK
            if token != semicolonsym
                error
            get next token
            emit RTN

STATEMENT
        if token == identsym
            symIdx = FINDSYMBOL (token, kind 2)
            if symIdx == -1
                if FINDSYMBOL (token, 1) != FINDSYMBOL (token, 3)
                    error
                else
                    error
            get next token
            if token != assignsym
                error
            get next token
            EXPRESSION
            emit STO (L = level – table[symIdx].level, M = table[symIdx].addr)
            return
        if token == beginsym
            do
                get next token
                STATEMENT
            while token == semicolonsym
            if token != endsym
                if token == identsym, beginsym, ifsym, whilesym, readsym, writesym,or
callsym
                    error
                else
                    error
            get next token
            return
        if token == ifsym
            get next token
            CONDITION
            jpcIdx = current code index
            emit JPC
            if token != thensym
                error
            get next token
            STATEMENT
            if token == elsesym
                jmpIdx = current code index
                emit JMP
```

```
        code[jpcIdx].m = current code index * 3
        STATEMENT
        code[jmpIdx].m = current code index * 3
    else
        code[jpcIdx].m = current code index * 3
    return
if token == whilesym
    get next token
    loopIdx = current code index
    CONDITION
    if token != dosym
        error
    get next token
    jpcIdx = current code index
    emit JPC
    STATEMENT
    emit JMP M = loopIdx * 3
    code[jpcIdx].m = current code index * 3
    return
if token == readsym
    get next token
    if token != identsym
        error
    symIdx = FINDSYMBOL (token, kind 2)
    if symIdx == -1
        if FINDSYMBOL (token ,1) != FINDSYMBOL(token, 3)
            error
        else
            error
    get next token
    emit READ
    emit STO (L = level – table[symIdx].level, M = table[symIdx].addr)
    return
if token == writesym
    get next token
    EXPRESSION
    emit WRITE
    return
if token == callsym
    get next token
    symIdx = FINDSYMBOL (token, kind 3)
    if symIdx == -1
        if FINDSYMBOL (token, 1) != FINDSYMBOL(token, 2)
            error
        else
            error
    get next token
    emit CAL (L = level – table[symIdx].level, symIdx)
```

```
CONDITION
	if token == oddsym
		get next token
		EXPRESSION
		emit ODD
	else
		EXPRESSION
		if token == eqlsym
			get next token
			EXPRESSION
			emit EQL
		else if token == neqsym
			get next token
			EXPRESSION
			emit NEQ
		else if token == lsssym
			get next token
			EXPRESSION
			emit LSS
		else if token == leqsym
			get next token
			EXPRESSION
			emit LEQ
		else if token == gtrsym
			get next token
			EXPRESSION
			emit GTR
		else if token == geqsym
			get next token
			EXPRESSION
			emit GEQ
		else
			error

EXPRESSION
	if token == subsym
		get next token
		TERM
		emit NEG
		while token == addsym || token == subsym
			if token == addsym
				get next token
				TERM
				emit ADD
			else
				get next token
				TERM
```

```
                        emit SUB
        else
                if token == addsym
                        get next token
                TERM
                while token == addsym || token == subsym
                        if token == addsym
                                get next token
                                TERM
                                emit ADD
                        else
                                get next token
                                TERM
                                emit SUB
        if token == ( identifier number odd
                error

TERM
        FACTOR
        while token == multsym || token == divsym || token == modsym
                if token == multsym
                        get next token
                        FACTOR
                        emit MUL
                else if token == divsym
                        get next token
                        FACTOR
                        emit DIV
                else
                        get next token
                        FACTOR
                        emit MOD

FACTOR
        if token == identsym
                symIdx_var = FINDSYMBOL (token, 2)
                symIdx_const = FINDSYMBOL(token, 1)
                if symIdx_var == -1 && symIdx_const == -1
                        if FINDSYMBOL(token, 3) != -1
                                error
                        else
                                error
                if symIdx_var == -1 (const)
                        emit LIT M = table[symIdx_const].val
                else if symIdx_const == -1 || table[symIdx_var].level > table[symIdx_const].level
                        emit LOD(L = level–table[symIdx_var].level, M = table[symIdx_var].addr)
                else
                        emit LIT M = table[symIdx_const].val
```

```
        get next token
else if token == numbersym
        emit LIT
        get next token
else if token == lparentsym
        get next token
        EXPRESSION
        if token != rparentsym
                error
        get next token
else
        error
```

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.
- 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 function should do a linear pass through the symbol table looking for the symbol name 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
- How does FINDSYMBOL work?

    - 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 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.

```
typedef struct
  {
        int kind;             // const = 1, var = 2
        char name[12];        // name up to 11 chars
        int val;              // number
        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.