University of Central Florida
Department of Computer Science
COP 3402: System Software
Montagne, Spring 2022

Homework #3 (Parser- Code Generator)

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

Objective:
In this assignment, you must implement a Recursive Descent Parser and Intermediate Code Generator for PL/0.

Component Descriptions:
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 won't reach 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 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

Notes on Implementation
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.

Error Handling

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. Please don't compress your files
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.
5. Include comments in your program

Rubric

10 Compiles

20 Produces some instructions before segfaulting or looping infinitely, not necessarily correct, but enough to demonstrate that your program is doing something.

15 Symbol Table

23 Errors are implemented correctly

10 If and While Structures

10 Expression, Term, and Factor

6 Read, Write, and Assignment Statements

6 Scope and Program

Appendix A: Example

Input

```
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 -a -s > output.txt

Output

```
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
Line OP Code   OP Name   L    M
0    7    JMP  0    9
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    RET  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    1
20   9    HAL  0    3
```

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 four 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; errors 18 and 19 are this type
D. Register Overflow Error - more register space is used than is available. Error 20 is 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 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
20. Register Overflow Error - found when a load instruction (LOD, LIT, RED) should be emitted, but there is no more space on the register stack. Keep track of space on the register stack by using a counter variable. It should be incremented when a load instruction is emitted and decremented when STO, ADD, SUB, MUL, DIV, EQL, NEQ, LSS, LEQ, GTR, GEQ, or JPC are emitted. The maximum register size is 10.

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 main code starts)
        add to symbol table (kind 3, "main", 0, level = 0, 0, unmarked)
        level = -1
        BLOCK
        emit HALT
        now we know where our procedures start, we saved those addresses in their addr field in the
                symbol table, so we can fix the M values of the CAL instructions and the M value
                of the initial JMP instruction

BLOCK
        Increment level
        CONST-DECLARATION
        x = VAR-DECLARATION (save how many variable spaces we need in the AR)
        PROCEDURE-DECLARATION
        we're about to start emitting code for the current procedure. Note the code address in the
                procedure's addr field on the symbol table so we can use it for CALs
        emit INC (M = x + 3)
        STATEMENT
        MARK
        Decrement level

CONST-DECLARATION

VAR-DECLARATION
        should return the number of variables declared so we know how many spaces to reserve in
                the AR
        when we add variables to the symbol table, the addr field should be equal to the number of
                variables before the current one being declared + 3
                so the first variable of a procedure should have addr = 0 + 3, the second should
                        have addr = 1 + 3, and so on

PROCEDURE-DECLARATION
        For each procedure:
                1 - process the procedure declaration
                2 - add it to the symbol table
                3 - call BLOCK
                4 - emit RET
```

```
STATEMENT
        assignment statements
                make sure the symbol is a variable available in the symbol table
                EXPRESSION
                emit STO (L = level – table[symIdx].level, M = table[symIdx].addr)
        begin statements
                do
                        get next token
                        STATEMENT
                while token == semicolonsym
        if statements
                CONDITION
                jpcIdx = current code index
                emit JPC (M = 0, because we don't know where we're jumping to yet: either after
                                the if structure or to the else section)
                STATEMENT
                if token == elsesym
                        jmpIdx = current code index
                        emit JMP (M = 0, because we don't know how long the else section will be
                                        yet)
                        code[jpcIdx].m = current code index
                        STATEMENT
                        code[jmpIdx].m = current code index
                else
                        code[jpcIdx].m = current code index
        while statement
                loopIdx = current code index (because we need to do our condition every time we
                                                loop, so we need to know where to jump back to)
                CONDITION
                jpcIdx = current code index
                emit JPC (M = 0, because this jumps to the end of the loop, but we don't know
                                where that is yet)
                STATEMENT
                emit JMP M = loopIdx
                code[jpcIdx].m = current code index
        read statements
                make sure the symbol is a variable available in the symbol table
                emit READ
                emit STO (L = level – table[symIdx].level, M = table[symIdx].addr)
        write statements
                EXPRESSION
                emit WRITE
        call statements
                make sure the symbol is 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 symbol in the symbol table so when we go back to
                                fix the CALs we know which procedure it's referencing)

CONDITION
```

```
        EXPRESSION
        if token == eqlsym
                EXPRESSION
                emit EQL
        else if token == neqsym
                EXPRESSION
                emit NEQ
        else if token == lsssym
                EXPRESSION
                emit LSS
        else if token == leqsym
                EXPRESSION
                emit LEQ
        else if token == gtrsym
                EXPRESSION
                emit GTR
        else if token == geqsym
                EXPRESSION
                emit GEQ

EXPRESSION
        if token == subsym
                TERM
                emit NEG
                while token == plussym || token == minussym
                        if token == plussym
                                TERM
                                emit ADD
                        else
                                TERM
                                emit SUB
        else
                if token == plussym
                        get next token
                TERM
                while token == plussym || token == minussym
                        if token == plussym
                                TERM
                                emit ADD
                        else
                                TERM
                                emit SUB

TERM
        FACTOR
        while token == multsym || token == divsym
                if token == multsym
                        FACTOR
                        emit MUL
                else
```

```
                    FACTOR
                    emit DIV

FACTOR
    if token == identsym
        find the symbol in the table, we want an available symbol (unmarked), it can be a
            a var or a const, but we want to prioritize a lower level. There are two
            symbols with the desired name, one const and one var, pick the one with
            the lower lex level.
        once you find the correct symbol,
        if it's a constant
            emit LIT M = table[symIdx].val
        else if it's 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 leave 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 skeleton
- 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 you can use in CONST-DECLARATION, VAR-DECLARATION, and PROCEDURE-DECLARATION to see if the name has already been used.
    - This function does 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 is a function given in the skeleton you can use in STATEMENT and FACTOR in order to find a symbol table entry. It returns -1 if one can't be found, the index 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 a function given in the skeleton you can use in BLOCK to mark the procedure's symbols after you're done with it.
    - 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 = 20];
```

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.