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

Homework #3 Parser & Code Generator

MAKE SURE YOU HAVE READ THE PROJECT OVERVIEW, ESPECIALLY THE PROJECT INSTRUCTIONS AND GUIDELINES SECTION

Goal:

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 Lexical Analyzer (HW2) and parses the tokens. It must be capable of reading in the tokens produced by your Lexical Analyzer (HW2) and print, if the program does not follow the grammar, a message indicating the type of error present **(reminder: if the lexical analyzer detects an error the compilation process stops before reaching the parser)**. A list of the errors that must be considered can be found in Appendix B. In addition, the Parser must fill out the Symbol Table, which contains all of the variables, procedure and array names within the PL/0 program. See Appendix D 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). The parser and intermediate code generator can be implemented individually or combined. The pseudocode we provide has them combined.

Error Handling

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

For this assignment, you must implement the function instruction *parse(lexeme *list) in parser.c. The Project Overview contains an explanation of testing and the skeleton. You may add as many helper functions and global variables as you desire. This function takes the lexeme list should return the assembly code unless there is an error, in which case it should return NULL. The primary function marks the end of the code before returning. This is necessary for the vm to work and should be preserved. parser.c also includes three printing functions: printerror which will print the error

message and free the code and symbol table arrays, printsymboltable which will print the symbol table, and printcode which will print the assembly code. Make sure to call one of these functions before you return from the parser function. For these functions to work, you must have the table, table index, code and code index as global variables. We also provide compiler.h which includes the instruction and symbol structs.

Grammar (EBNF) of tiny PL/0:

program ::= block "**.**" **.**
block ::= var-declaration procedure-declaration statement**.**
var-declaration ::= ["**var**" ident ["**[**" number "**]**"] {"**,**" ident ["**[**" number "**]**"]} "**;**"]**.**
procedure-declaration ::= { "**procedure**" ident "**;**" block "**;**" }.
statement ::= [ident ["**[**" expression "**]**"] "**:=**" expression
| "**call**" ident
| "**begin**" statement { "**;**" statement } "**end**"
| "**if**" condition "**?**" statement ["**:**" statement]
| "**do**" statement "**while**" condition
| "**read**" ident ["**[**" expression "**]**"]
| "**write**" expression
| **ε**] **.**
condition ::= expression rel-op expression**.**
rel-op ::= "**==**"|"**<>**"|"**<**"|"**<=**"|"**>**"|"**>=**"**.**
expression ::= ["**-**"] term { ("**+**"|"**-**") term}**.**
term ::= factor {("*****"|"**/**"|"**%**") factor}**.**
factor ::= ident ["**[**" expression "**]**"] | 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 the .o files 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.

Submission Instructions:

Submit via WebCourses:

1. Source code named parser.c. 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. Instructions to use the program and author names in a readme text file. (If you didn't alter the package structure, your readme will just be author names)
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. (If your partner receives a grade but you don't, just send us a message/email)
4. Include comments in your program.
5. Output should print to the screen and should follow the format in the examples. If your output does not follow this format, you will experience a delay in grading.

Rubric

15	Compiles
10	Produces some instructions (which clearly change depending on the input file) before segfaulting, looping infinitely, or erroring on all inputs
24	Errors
10	Symbol Table
19	Statements
9	Procedures
13	Expression, Term, Factor

Appendix A: Example

If the input is:

```
var a, b, z[2], y[3];
procedure J;;
begin
     call J;
     if 4 / 2 == 1 ?
          begin
               read a;
               read y[2];
               read y[3/3];
               write 8;
               write 8 * 2;
          end;
     a := 3;
     y[0] := 1;
     z[0+1] := 2;

     do
          begin
               a := y[0] * y[0+1] + b;
               b := 8 % 4 + (-1 + 5);
          end
     while 3 <> 3;
end.
```

The command is "./a.out statements.txt -s -a"

The output should be:

```
Symbol Table:
Kind | Name        | Size | Level | Address | Mark
---------------------------------------------------
   3 |        main |     0 |     0 |     3 |     1
   1 |           a |     0 |     0 |     3 |     1
   1 |           b |     0 |     0 |     4 |     1
   2 |           z |     2 |     0 |     5 |     1
   2 |           y |     3 |     0 |     7 |     1
   3 |           J |     0 |     0 |     1 |     1
Line OP Code   OP Name   R    L    M
0    7    JMP  0    0    3
1    6    INC  0    0    3
2    2    RET  0    0    0
3    6    INC  0    0    10
```

```
4     5     CAL   0     0     1
5     1     LIT   0     0     4
6     1     LIT   1     0     2
7     16    DIV   0     0     1
8     1     LIT   1     0     1
9     18    EQL   0     0     1
10    8     JPC   0     0     32
11    1     LIT   0     0     3
12    10    RED   1     0     0
13    4     STO   1     0     0
14    1     LIT   0     0     2
15    1     LIT   1     0     7
16    13    ADD   0     0     1
17    10    RED   1     0     0
18    4     STO   1     0     0
19    1     LIT   0     0     3
20    1     LIT   1     0     3
21    16    DIV   0     0     1
22    1     LIT   1     0     7
23    13    ADD   0     0     1
24    10    RED   1     0     0
25    4     STO   1     0     0
26    1     LIT   0     0     8
27    9     WRT   0     0     0
28    1     LIT   0     0     8
29    1     LIT   1     0     2
30    15    MUL   0     0     1
31    9     WRT   0     0     0
32    1     LIT   0     0     3
33    1     LIT   1     0     3
34    4     STO   1     0     0
35    1     LIT   0     0     0
36    1     LIT   1     0     7
37    13    ADD   0     0     1
38    1     LIT   1     0     1
39    4     STO   1     0     0
40    1     LIT   0     0     0
41    1     LIT   1     0     1
42    13    ADD   0     0     1
43    1     LIT   1     0     5
44    13    ADD   0     0     1
45    1     LIT   1     0     2
46    4     STO   1     0     0
```

```
47   1    LIT  0    0    3
48   1    LIT  1    0    0
49   1    LIT  2    0    7
50   13   ADD  1    1    2
51   3    LOD  1    0    1
52   1    LIT  2    0    0
53   1    LIT  3    0    1
54   13   ADD  2    2    3
55   1    LIT  3    0    7
56   13   ADD  2    2    3
57   3    LOD  2    0    2
58   15   MUL  1    1    2
59   1    LIT  2    0    4
60   3    LOD  2    0    2
61   13   ADD  1    1    2
62   4    STO  1    0    0
63   1    LIT  0    0    4
64   1    LIT  1    0    8
65   1    LIT  2    0    4
66   17   MOD  1    1    2
67   1    LIT  2    0    1
68   12   NEG  2    0    2
69   1    LIT  3    0    5
70   13   ADD  2    2    3
71   13   ADD  1    1    2
72   4    STO  1    0    0
73   1    LIT  0    0    3
74   1    LIT  1    0    3
75   19   NEQ  0    0    1
76   1    LIT  1    0    0
77   18   EQL  0    0    1
78   8    JPC  0    0    47
79   11   HLT  0    0    0
```

Appendix B: Error Messages

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. Symbol names must be identifiers – found when the flow of control is in **var-declaration** and ident is missing after "var" or "," OR in **procedure-declaration** when an ident is missing after "procedure"
3. 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
4. Array sizes must be given as a single, nonzero number – found when the flow of control is in **var-declaration** and an array has been found (lbracketsym present), but there is no number for size OR the programmer has tried to give size as an expression (so the number is followed by "*", "/", "%", "+", or "-") OR the number present is 0
5. [must be followed by] - found in **var-declaration,** assignment and read cases of **statement**, or in **factor** when] is missing (with the additional condition in **var-declaration** that error 4 was not present)
6. Multiple symbols in var declaration must be separated by commas – found in **var-declaration** when you check for the ending semicolon and find an identifier instead
7. Symbol declarations should close with a semicolon – found in **var-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
8. Procedure declarations should contain a semicolon before the body of the procedure begins – found when the flow of control is in **procedure-declaration** and ; is missing after the identifier
9. Procedures may not be assigned to, read, or used in arithmetic – found in **statement** in the read case when the identifier is a procedure OR the identifier present is a procedure (has kind 3) and in the assignment case when the identifier is a procedure, also found in **factor** when the identifier is a procedure
10. 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
11. Variables cannot be indexed - found in **statement** in the assignment and read cases when there is a left bracket, but no valid array symbol OR in **factor** in the identifier case when there is a left bracket, but no valid array symbol
12. Arrays must be indexed - found in **statement** in the assignment and read cases when there is not a left bracket and no valid array symbol OR in **factor** in the identifier case when there is not a left bracket and no valid array symbol

13. Assignment operator missing - found in **statement** in the assignment case when := is missing
14. Register Overflow Error - found when a load instruction (LOD, LIT, RED) should be emitted, but there is no more space in the register file. Keep track of space in the register file by using a counter variable. It should be incremented when a load instruction is emitted and decremented when ADD, SUB, MUL, DIV, MOD, EQL, NEQ, LSS, LEQ, GTR, GEQ, or JPC are emitted. It should be decremented twice when STO is emitted. The maximum register size is 10.
15. 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)
16. 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, call, begin, if, do, read, or write
17. begin must be followed by end – found in **statement** when the end symbol is expected and the symbol present is neither end, identifier, call, begin, if, do, read, nor write
18. if must be followed by ? – found in **statement** in the if case when flow of control returns from **condition** and the current symbol is not ?
19. do must be followed by while – found in **statement** in the do-while case when flow of control returns from **statement** and the current symbol is not while
20. read must be followed by a var or array identifier – found in **statement** in the read case when the identifier is missing OR the identifier present is a procedure (has kind 3)
21. 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
22. 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
23. (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
24. Arithmetic expressions may only contain arithmetic operators, numbers, parentheses, 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)

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 C: Pseudocode

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 (in Project Overview), 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
        note where the procedure is in the symbol table
        x = VAR-DECLARATION (save how many spaces we need in memory)
        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 + 1)
        STATEMENT
        MARK
        Decrement level

VAR-DECLARATION
        should return the number of memory spaces declared (1 for each var + the size of each
                array) for the INC of the procedure
        when we add vars and arrays to the symbol table, the addr field should be equal to the
                number of memory spaces that were declared before the var
                so the first symbol will have addr = 0
                        if it was a var, the next symbol will have addr = 1
                        if it was an array of size 5, the next symbol will have addr = 5

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
                see note about load and store in FAQ
        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)
        begin statements
                do
                        get next token
                        STATEMENT
                while token == semicolonsym
        if statements
                CONDITION
                jpcIdx = current code index
                emit JPC (R= register counter, 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 == colonsym
                        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
        do-while statement
                loopIdx = current code index (because we need to know where the action starts so
                                we can come back to it later if the condition is true)
                STATEMENT
                CONDITION
                now we need a JPC to see if we should do another iteration, but JPC only activates
                        if the condition code generated a false result. So we need to reverse the
                        condition result
                emit LIT (M = 0)
                emit EQL
                emit JPC (M = loopIdx from earlier)
        read statements
                see note about load and store in FAQ
        write statements
                EXPRESSION
                emit WRT

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 == addsym || token == subsym
                        if token == addsym
                                TERM
                                emit ADD
                        else
                                TERM
                                emit SUB
        else
                TERM
                while token == addsym || token == subsym
                        if token == addym
                                TERM
                                emit ADD
                        else
                                TERM
                                emit SUB

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

```
FACTOR
    if token == identsym
        see note about load and store in FAQ
    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 11 0 0 0 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 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
- How do I know which register my instructions should point to?
    - Use a register_counter variable. Start at -1 and increment whenever you emit LIT. (You only emit LOD in one place and it actually uses a LIT for the memory location so it will load to the same register as the LIT so you don't need to also increment for LOD). Decrement whenever you emit (a logic instruction (EQL, NEQ, LSS, etc.), an arithmetic instruction (ADD, SUB, MUL, DIV, MOD), or a JPC. Don't decrement for negate instructions, these should store the result in the same register being acted on. For STO instructions, you have to decrement the counter by 2 because STO empties the register holding the value being stored AND the register holding the memory location.
    - For LOD and STO, see the next note on R values.
    - For LIT, NEG, and JPC which only use one register, just use register_counter.
    - For all the other instructions: R is the destination register, L and M are the source registers. L is always the one on the left in the expression and M is the one on the right (ex SUB R = L - M). Your register_counter variable will be pointing to the desired M register and the desired L register is one less. Use the L register for the destination as well. (ex. If register counter = 3, Reg[2] = Reg[2] + Reg[3], emit ADD (op, 2, 2, 3))
- Load and Store Note
    - Arrays make load and store complicated, but thankfully all three instances (assignment and read cases of STATEMENT and identifier case of FACTOR) use very similar logic.
        1. Check for a valid var symbol or a valid array symbol
        2. If there isn't a var or an array, there's an error (error 9 or error 10, the presence of a valid procedure determines which one)

3. Check for a left bracket.
    1. Make sure they're not trying to index a variable (is there a valid array symbol)
    2. Call expression to generate the code to calculate the array index
    3. The register use counter will be pointing to the register the index is stored in. Save this.
    4. Check for the right bracket.
    5. **For assignment case**: check for the assignment operator, then call expression to generate code for the calculation whose result is being saved at the array index || **For read case**: emit the RED || **For factor case:** do nothing
    6. We need to finish calculating the memory location, it's the array index plus the memory location of the first array element (table[symbol].addr). Load the first array element memory location as a literal (emit LIT (R = register_counter)). Then add that register with the array index we calculated earlier (emit ADD (R = array index register, L = array index register, M = register_counter))
    7. Emit the LOD or STO
        1. STO, R = register_counter [currently pointing to the register with the result of the expression being stored || the read result], L = level - table[array].level, M = memory location register
        2. LOD, R = register_counter [currently pointing to the register with the result of the memory address], L = level - table[array].level, M = memory address register [M and R will be the same]
4. If there isn't a left bracket, check that there is a valid var symbol
    1. We need to load the var's memory address into a literal and save the register number (register_counter)
    2. **For assignment case**: check for the assignment operator, then call expression to generate code for the calculation whose result is being saved at the array index || **For read case**: emit the RED || **For factor case:** do nothing
    3. Emit the LOD or STO
        1. STO, R = register_counter [currently pointing to the register with the result of the expression being stored || the read result], L = level - table[var].level, M = memory location register

2. LOD, R = register_counter [currently pointing to the register with the result of the memory address], L = level - table[array].level, M = memory address register [M and R will be the same]

Appendix E:

Symbol Table

Recommended data structure for the symbol.

```
typedef struct
   {
        int kind;              // var = 1, array = 2, procedures = 3
        char name[12];         // name up to 11 chars
        int size;              // length of the array
        int level;             // lexical level
        int addr;              // var = memory address, array = memory address of first
                                       element, procedure = symbol table index when the
                                       procedure is added to the symbol table, address of
                                       first instruction at time of printing symbol table
        int mark;              // indicates whether the symbol is active
   } symbol;

symbol_table[MAX_SYMBOL_TABLE_SIZE = 20];
```

For variables, you must store kind, name, level, addr, and mark.
For arrays, you must store kind, name, size, 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. When symbols are added to the symbol table they should be marked. The MARK function called at the end of BLOCK will mark the symbols. By the end of the program when you print the symbol table, all symbols should be marked..