University of Central Florida
Department of Computer Science
COP 3402: System Software
Spring 2023

Homework #3 (Tiny PL/0 compiler)

Due 3/8/2023 by 11:59 p.m.

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

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

Make a copy of lex.c

In the new file lex.c, apply the following changes:

The token list, output HW2, must be kept in the program and or written out to a file(this option will make the parser/codegen slower).

Rename the name of the new copy of lex.c as parsercodegen.c.

Implement the parser/code generator in this file called parsercodegen.c, this means that you will continue inserting code in parsercodegen.c

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

Example of a program written in PL/0:

```
var x, y;
begin
        x := y * 2;
end.
```

Component Descriptions:

The **parser/codegen** must be capable of getting 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 **(This time: if the scanner step detects an error the compilation process must stop and the error must be indicated**, **similarly in the parser step, if a syntax error is detected, 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 populate the Symbol Table, which contains all of the variables 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, and code for the virtual machine (HW1) will be generated.

Include always as first entry in the symbol table:

```
Kind | Name         | Value | Level | Address | Mark
---------------------------------------------------
   3 |         main |     0 |     0 |     3   |  0
```

For HW3, we will select teams at random to review the compiler. Each team member must know how the compiler and the vm work. If any team member fails in answering a question, a penalty of (-10) will be applied to the whole team in HW3.

Submission Instructions:

1.- Submit via WebCourses:

1. Source code of the tiny- PL/0 compiler (parsercodegen.c).
2. A text file with instructions on how to use your program (readme.txt.).
3. As many Input and output files (cases) to show each one of the errors your compiler can detect, and one correct program. Name them errorin1.text, errorout1.text, errorin2.text, errorout2.text, and so on.
4. All files should be compressed into a single .zip format.
5. Late policy is the same as HW1 and HW2.
6. Only one submission per team: the name of all team members must be written in the source code header file and in the readme document.
7. Include comments in your program
8. Output should print to the screen and should follow the format in Appendix A. A deduction of 5 points will be applied to submissions that do not print to the screen.
9. The input file should be given as a command line argument. A deduction of 5 points will be applied to submissions that do not implement this.

Error Handling

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

Output specifications:

- If you find an error, print it to the screen using the format "Error : <error message>"
- Otherwise, print the assembly code for the virtual machine (HW1) and the symbol table.

See Appendix A

Rubric

15 – Compiles
20 – Produces some instructions before segfaulting or looping infinitely
5 – Follows IO specifications (takes command line argument for input file name and prints output to console)
5 – README.txt containing author names
10 – Correctly create symbol table
10 – Correctly implements expression, term, and factor
10 – Loads and store values correctly
5 – Supports error handling
10 – Correctly implements if statements
10 – Correctly implements while statements

******* If a program does not compile, your grade is zero.**

******* If you do not follow the specifications, your grade is zero. For instance, implementing programming constructs not present in the PL/0 grammar. For example, if you implement procedures, procedure call, it-then-else, your grade will be zero**

Appendix A:

Traces of Execution:

Example 1, if the input is:

```
var x, y;
begin
     x := y * 2;
end.
```

The output should look like:

Assembly Code:

```
Line     OP     L     M
   0     JMP    0     3
   1     INC    0     5
   2     LOD    0     4
   3     LIT    0     2
   4     OPR    0     4
   5     STO    0     3
   6     SYS    0     3
```

Symbol Table:

```
Kind | Name         | Value | Level | Address | Mark
---------------------------------------------------
   3 |         main |     0 |     0 |     3 |     1
   1 |            x |     0 |     0 |     3 |     1
   2 |            y |     0 |     0 |     4 |     1
```

Example 2, if the input is:

```
var x, y;
begin
     z:= y * 2;
end.
```

The output should look like:

```
Error: undeclared identifier z
```

Appendix B:

EBNF of tiny PL/0:

program ::= block "**.**" .
block ::= const-declaration var-declaration statement.
constdeclaration ::= ["**const**" ident "**=**" number {"**,**" ident "**=**" number} "**;**"].
var-declaration ::= ["**var**" ident {"**,**" ident} "**;**"].
statement ::= [ident "**:=**" expression
| "**begin**" statement { "**;**" statement } "**end**"
| "**if**" condition "**then**" statement
| "**while**" condition "**do**" statement
| "**read**" ident
| "**write**" expression
| **empty**] .
condition ::= "**odd**" expression
| expression rel-op expression.
rel-op ::= "**=**"|"**<>**"|"**<**"|"**<=**"|"**>**"|"**>=**".
expression ::= ["**+**"|"**-**"] term { ("**+**"|"**-**") term}.
term ::= factor {("*****"|"**/**") factor}.
factor ::= ident | number | "**(**" expression "**)**".
number ::= digit {digit}.
ident ::= letter {letter | digit}.
digit ;;= "**0**" | "**1**" | "**2**" | "**3**" | "**4**" | "**5**" | "**6**" | "**7**" | "**8**" | "**9**".
letter ::= "**a**" | "**b**" | ... | "**y**" | "**z**" | "**A**" | "**B**" | ... |"**Y**" | "**Z**".

Based on Wirth's definition for EBNF we have the following rule:
[ ] means an optional item.
{ } means repeat 0 or more times.
Terminal symbols are enclosed in quote marks.
A period is used to indicate the end of the definition of a syntactic class.

Appendix C:

Error messages for the tiny PL/0 Parser:

- program must end with period
- const, var, and read keywords must be followed by identifier
- symbol name has already been declared
- constants must be assigned with =
- constants must be assigned an integer value
- constant and variable declarations must be followed by a semicolon
- undeclared identifier
- only variable values may be altered
- assignment statements must use :=
- begin must be followed by end
- if must be followed by then
- while must be followed by do
- condition must contain comparison operator
- right parenthesis must follow left parenthesis
- arithmetic equations must contain operands, parentheses, numbers, or symbols

These are all the error messages you should handle in your parser.

Appendix D: Pseudocode

```
SYMBOLTABLECHECK (string)
        linear search through symbol table looking at name
        return index if found, -1 if not

PROGRAM
        BLOCK
        if token != periodsym
                error
        emit HALT

BLOCK
        CONST-DECLARATION
        numVars = VAR-DECLARATION
        emit INC (M = 3 + numVars)
        STATEMENT

CONST-DECLARATION
        if token == const
                do
                        get next token
                        if token != identsym
                                error
                        if SYMBOLTABLECHECK (token) != -1
                                error
                        save ident name
                        get next token
                        if token != eqlsym
                                error
                        get next token
                        if token != numbersym
                                error
                        add to symbol table (kind 1, saved name, number, 0, 0)
                        get next token
                while token == commasym
                if token != semicolonsym
                        error
                get next token

VAR-DECLARATION – returns number of variables
        numVars = 0
        if token == varsym
                do
                        numVars++
                        get next token
```

```
                    if token != identsym
                            error
                    if SYMBOLTABLECHECK (token) != -1
                            error
                    add to symbol table (kind 2, ident, 0, 0, var# + 2)
                    get next token
            while token == commasym
            if token != semicolonsym
                    error
            get next token
    return numVars

STATEMENT
    if token == identsym
            symIdx = SYMBOLTABLECHECK (token)
            if symIdx == -1
                    error
            if table[symIdx].kind != 2 (not a var)
                    error
            get next token
            if token != becomessym
                    error
            get next token
            EXPRESSION
            emit STO (M = table[symIdx].addr)
            return
    if token == beginsym
            do
                    get next token
                    STATEMENT
            while token == semicolonsym
            if token != endsym
                    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
            code[jpcIdx].M = current code index
            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)
                code[jpcIdx].M = current code index
                return
        if token == readsym
                get next token
                if token != identsym
                        error
                symIdx = SYMBOLTABLECHECK (token)
                if symIdx == -1
                        error
                if table[symIdx].kind != 2 (not a var)
                        error
                get next token
                emit READ
                emit STO (M = table[symIdx].addr)
                return
        if token == writesym
                get next token
                EXPRESSION
                emit WRITE
                return

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 == lessym
```

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

TERM
```

```
        FACTOR
        while token == multsym || token == slashsym || token == modsym
                if token == multsym
                        get next token
                        FACTOR
                        emit MUL
                else if token == slashsym
                        get next token
                        FACTOR
                        emit DIV
                else
                        get next token
                        FACTOR
                        emit MOD

FACTOR
        if token == identsym
                symIdx = SYMBOLTABLECHECK (token)
                if symIdx == -1
                        error
                if table[symIdx].kind == 1 (const)
                        emit LIT (M = table[symIdx].Value)
                else (var)
                        emit LOD (M = table[symIdx].addr)
                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
```

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
  } symbol;

symbol_table[MAX_SYMBOL_TABLE_SIZE = 500];
```

For constants, you must store kind, name and value.
For variables, you must store kind, name, L and M.