Homework #4 (PL/0 Compiler and VM)

COP 3402: Systems Software

Summer 2025

Due Date: **[July 20th 2025 by 11:59 p.m.]**

> **Disclaimer:** This document may not cover all possible scenarios - when in doubt, ask the instructor or a TA.
>
> All official updates (test cases, clarifications, etc.) will be posted as **Canvas Announcements**.
>
> **Check announcements regularly** for critical updates.

Assignment Overview

For this assignment, you will extend your existing **PL/0** compiler (from HW3) to:

1. Implement new grammar elements involving `procedure`, `call`, and `else` clause in `if` statements.
2. Update your **HW1 Virtual Machine (VM)** to recognize and execute the modulo (`mod`) instruction (`OPR 0 11`).
3. Generate an executable file named `elf.txt`, which will serve as the input for your **updated VM**.
4. Demonstrate correct parsing, symbol table management, and code generation for these extended grammar features.

Objective

You must create a **Recursive Descent Parser and Code Generator** for the extended PL/0 language. In particular:

- Procedures must be supported via `procedure-declaration` and the `call` statement.
- `if` statements must include an `else` part, ensuring complete code generation paths in both branches.

- The `mod` operator must be recognized in the grammar and translated to `2 0 11`.
- The `elf.txt` output should be directly usable by your updated VM from HW1.

Component Descriptions

Your compiler must:

- Accept PL/0 source code from a file specified on the command line (e.g., `./a.out input.txt`).
- Detect both lexical and syntactical errors, halting with a clear message if any occur.
- Generate correct code for the extended language constructs, including `procedure` declarations, `call` statements, and `else` block in `if` statements.
- Build and maintain a symbol table that supports multiple lexical levels (for nested procedures).

Submission Requirements

I. Essential Files

Submit the following in a single `submission.zip` archive via WebCourses:

- `hw4compiler.c`: Your primary compiler source file.
- `vm.c`: Your updated VM from HW1, now capable of handling the modulo instruction (`2 0 11`).
- Any additional files needed to compile your `hw4compiler.c` and `vm.c`.
- `README.md`: Instructions on how to compile and run your compiler *and* VM.
- **One sample input file** (correct PL/0 program) plus its corresponding **elf.txt** output file.
- **A folder of test cases**: At least two distinct test cases demonstrating `procedure` and `call` usage, plus error scenarios.

II. Formatting and Delivery

- Your compiler must read the input filename from the command line.
- Your compiler must display generated instructions in the terminal in the format:

  ```
  OPcode L M
  ```

  for example:

  ```
  LIT 0 10
  ```

- Your compiler must also create an `elf.txt` file containing machine codes for the VM to execute. for example:

  ```
  1 0 10
  ```

- The symbol table must be displayed in the terminal in the format:

  ```
  Kind | Name | Value | Level | Address | Mark
  ```

III. Additional Guidelines

- Include clear comments throughout your code
- List all team members in both README.md and source code header
- Only one team submission will be graded.
- Your submission must compile and run on **Eustis**. If it does not compile, the score is **0**.
- No late submissions will be accepted after two days past the deadline.
- Ensure your submission contains all necessary files to compile and run on Eustis

Error Handling

- If your compiler encounters an error (lexical, syntactic, or semantic), it should:
  (a) Print a concise error message to the terminal (e.g., `Error: undeclared identifier x`).
  (b) Halt further compilation immediately.
- Inherit the error types and messages from HW2 and HW3.

Output Specification

When Errors Occur

If your compiler identifies any error, it should:

```
Error : <error message >
```

When No Errors Are Found

If the source program is valid:

1. Display the input PL/0 program in the terminal.
2. Display the message: `No errors, program is syntactically correct`.
3. Print the generated assembly instructions to the terminal using mnemonic OP codes (e.g., `JMP 0 30`) along with the line number and the complete symbol table.
4. Create a file named `elf.txt` containing the numeric op codes (e.g., `7 0 30`), which your VM can load and execute.

Important: The assembly code and symbol table must be printed directly to the terminal, not saved to a file. Only the `elf.txt` file should be created as a separate output file.

Specific Requirements

- **Modifying the Lexical Scanner**: Replace any legacy `oddsym` token with `modsym` = 1. **Notice that grammar is slightly different that the grammar of HW3.**
- **Procedure Declarations**: Must appear in the `procedure-declaration` section of the grammar.
- **Call Statement**: Invokes a procedure by its identifier.
- `if-else` **Clause**: The `if` statement must contain an `else` block before `fi`. Omission of `else` is not allowed.
- **First Instruction**: Must always be `JMP 0 <some_address>`.

Lexical Conventions

A numerical value is assigned to each token (internal representation) as follows:

```
modsym = 1, identsym = 2, numbersym = 3, plussym = 4, minussym = 5,
multsym = 6, slashsym = 7, fisym = 8, eqlsym = 9, neqsym = 10,
lessym = 11, leqsym = 12, gtrsym = 13, geqsym = 14, lparentsym = 15,
rparentsym = 16, commasym = 17, semicolonsym = 18, periodsym = 19,
becomessym = 20, beginsym = 21, endsym = 22, ifsym = 23, thensym = 24,
whilesym = 25, dosym = 26, callsym = 27, constsym = 28, varsym = 29,
procsym = 30, writesym = 31, readsym = 32, elsesym = 33.
```

Important: `modsym` is a new reserved symbol introduced in this homework with token value 1. It represents the modulo operator in the grammar.

VM Instruction for Modulus

The modulus operation should be implemented using `OPR 0 11` with the following behavior:

```
11  MOD     Modulus: pop two values from the stack, divide second by
            first
            and push the remainder
            pas[sp + 1] <- pas[sp + 1] % pas[sp]
```

Failure to implement the `mod` instruction exactly as `OPR 0 11` (or displayed as `MOD 0 11` in assembly) will result in an automatic zero.

Grading Rubric

Your assignment starts with a base score of 100 points. The following deductions apply:

Critical Errors (Automatic Zero)

- **-100 points:** Does not compile on Eustis. This includes:
  - Compilation errors preventing generation of the executable file
  - Program produces immediate segmentation fault
  - Program crashes while running grading test cases
- **-100 points:** Does not accept input filename from command line (e.g., `./a.out input_file.txt`)
- **-100 points:** Compiler follows a different grammar than specified in section Appendix B
- **-100 points:** Submitting HW3 again without implementing procedures and call
- **-100 points:** Using any method other than the marking algorithm for symbol table management. The implementation must explicitly use the mark column to track symbol availability. Alternative implementations will be considered plagiarism.

VM Implementation Issues

- **-30 points:** HW1's modified VM source code (`vm.c`) is not submitted
- **-30 points:** HW1's VM source code is not modified to support the modulo instruction as specified
- **-10 points:** HW1's VM does not support the same I/O specification as the compiler

Fundamental Implementation Issues

- **-80 points:** Compiles but does nothing
- **-70 points:** Produces some instructions before segfaulting or looping infinitely
- **-30 points:** Not implementing procedures in the "block" correctly
- **-30 points:** Not implementing call statements correctly
- **-15 points:** Incorrect implementation that generates wrong instructions for if-else statements

Error Handling and Code Generation

- **-10 points:** Not supporting error handling for procedures (including error messages)
- **-10 points:** Not supporting error handling for call (including error messages)
- **-10 points:** Does not generate the elf.txt executable file for the VM
- **-10 points:** Does not display the generated instructions in the terminal

Symbol Table and Scope Management

- **-10 points:** Program does not handle variables with the same name at different levels correctly
- **-10 points:** Level information not managed correctly (global environment should be level 0, with procedures incrementing accordingly)
- **-10 points:** Marking algorithm for symbol table management does not work correctly. Every symbol must have a mark of 0 upon initial insertion and a mark of 1 once they are no longer usable. This mark column must be explicitly used as the mechanism for tracking symbol availability.

Instruction Generation Accuracy

- **-5 points per occurrence:** JMP instruction's M, JPC instruction's M, or CAL instruction's M not fully divisible by 3 after the subtraction of 10.
- **-5 points per occurrence:** JMP, JPC, or CAL instruction not leading to the correct index in the code list

Documentation and Testing

- **-5 points:** No README.txt containing author names
- **-2.5 points:** No sample input file and sample output file
- **-2.5 points:** No test cases folder

Output Format and Compatibility

- **-5 points:** Significantly misaligned output format compared to the examples in section Appendix A for both correct and error cases. Your output must match overall structure shown in the examples, the spacing and alignment can be different as long as it is readable.
- **-30 points:** If your modified HW1 VM (`vm.c`) fails to correctly process the `elf.txt` file produced by your compiler. Your VM must be able to read and execute the compiler's output without errors.

Important Note: The grading team reserves the right to deduct additional points for serious issues not explicitly listed in this rubric. Therefore, please thoroughly test your code on the Eustis system before submission to ensure it meets all requirements and functions correctly across various test cases.

Note: Multiple deductions may apply to a single submission. The minimum score is 0.

Helpful Hints:

- Focus on implementing exactly what's in the grammar specification - no more, no less
- Remember that error detection should immediately halt processing
- Ensure your symbol table only contains entries that appear in the actual input
- Test your implementation against all error cases listed in section Appendix C
- Interpret the grammar carefully rather than following pseudocode examples blindly
- Double-check your implementation against the grammar rules before submission

Appendix A Traces of Execution

This appendix provides examples of expected input and output for your compiler.

Example 1

Input:

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

Expected Output in the terminal:

```
No errors, program is syntactically correct.

Assembly Code:

Line    OP    L    M
  0    JMP    0    13
  1    INC    0    5
  2    LOD    0    4
  3    LIT    0    2
  4    OPR    0    3
  5    STO    0    3
  6    SYS    0    3

Symbol Table:

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

Expected Output in the elf.txt file:

```
7 0 13
6 0 5
3 0 4
1 0 2
```

```
2 0 3
4 0 3
9 0 3
```

Example 2: Error Handling

Input with Error:

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

Expected Output:

```
Error: undeclared identifier z
```

There will be no elf.txt file generated for this example.

Appendix B Grammar Specification

The following grammar defines the syntax for the tiny PL/0 language that your compiler must implement:

```
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 "
            | statement " else " " while " condition
            | " do" statement
            | " read "    ident
            | " write "

condition ::= expression rel - op expression

expression ::= term {(" +" | " -") term }

term ::= factor {("*" | "/" | " mod ") factor }

factor ::= ident | number | "(" expression ")"

number ::= digit { digit }

ident ::= letter { letter | digit }

rel - op ::= "=" | " <>" | " <" | " <=" | " >" | " >="

digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

letter ::= " a" | " b" | ... | " y" | " z" | " A" | " B" | ... | " Y"
```

Appendix C Error Messages

The following are the required error messages that your parser must handle:

1. Use = instead of :=.
2. = must be followed by a number.
3. Identifier must be followed by =.
4. const, var, procedure must be followed by identifier.
5. Semicolon or comma missing.
6. Incorrect symbol after procedure declaration.
7. Statement expected.
8. Incorrect symbol after statement part in block.
9. Period expected.
10. Semicolon between statements missing.
11. Undeclared identifier.
12. Assignment to constant or procedure is not allowed.
13. Assignment operator expected.
14. call must be followed by an identifier.
15. Call of a constant or variable is meaningless.
16. then expected.
17. Semicolon or end expected.
18. do expected.
19. Incorrect symbol following statement.
20. Relational operator expected.
21. Expression must not contain a procedure identifier.
22. Right parenthesis missing.
23. The preceding factor cannot begin with this symbol.
24. An expression cannot begin with this symbol.
25. This number is too large.

26. Identifier too long.

27. Invalid symbol.

Important Implementation Notes:

- **Identifiers:** Maximum 11 characters.
- **Numbers:** Maximum 5 digits.
- **Invalid symbols:** Characters not in the PL/0 grammar (e.g., %) must be rejected.
- **Comments and whitespace:** Must be ignored and not tokenized.

Note: Not all of these error messages may be used in every implementation, and you may create additional error messages to more accurately represent certain situations. However, when applicable, you should use these standard error messages for consistency.

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
    num Vars = VAR -
    DECLARATION PROCEDURE -
    DECLARATION
    emit INC ( M = 3 + num Vars )
    STATEMENT

CONST - DECLARATION
    if token == constsym
        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
    num Vars = 0
    if token == varsym
        do
            num Vars ++
            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 num Vars

PROCEDURE - DECLARATION { Newly added }
    while token == procsym
        get next token
        if token != identsym
            error
        get next token
        if token != semicolonsym
            error
        get next token
        add to symbol table ( kind 3 , ident , level , code Index )
        BLOCK ( level + 1)
        if token != semicolonsym
            error
        get next token

STATEMENT
    if token == identsym
        sym Idx = SYMBOLTABLECHECK ( token )
        if sym Idx == -1
            error
        if table [ sym Idx ]. kind != 2 ( not a var )
            error
        get next token
        if token != becomessym
            error
        get next token
        EXPRESSION
        emit STO ( M = table [ sym Idx ]. addr )
        return
    if token == callsym { Newly added }
        get next token
        if token != identsym
            error
        idx = SYMBOLTABLECHECK ( token )
        if idx == -1
            error
        if symbol Table [ idx ]. kind != 3
            error
        emit CAL ( L, symbol Table [ idx ]. addr )
        get next token
        return
    if token == beginsym
        do
            get next token
            STATEMENT
        while token == semicolonsym
        if token != endsym
            error
```

```
        get next token
        return
    if token == ifsym { Needs modification }
        get next token
        CONDITION
        jpc Idx = current code index
        emit JPC
        if token != thensym
            error
        get next token
        STATEMENT
        code [ jpc Idx ]. M = current code
        index if token != fisym
            error
        get next token
        return
    if token == whilesym
        get next token
        loop Idx = current code
        index CONDITION
        if token != dosym
            error
        get next token
        jpc Idx = current code index
        emit JPC
        STATEMENT
        emit   JMP   ( M   =   loop Idx )
        code [ jpc Idx ]. M = current code
        index return
    if token == readsym
        get next token
        if token != identsym
            error
        sym Idx = SYMBOLTABLECHECK ( token )
        if sym Idx == -1
            error
        if table [ sym Idx ]. kind != 2 ( not a var )
            error
        get next token
        emit READ
        emit STO ( M = table [ sym Idx ]. addr )
        return
    if token == writesym
        get next token
        EXPRESSION
        emit WRITE
        return

CONDITION
    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
    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
        sym Idx = SYMBOLTABLECHECK ( token )
```

```
214         if sym Idx == -
215         1
216         if table [ sym Idx ]. kind == 1 ( const )
217             emit LIT ( M = table [ sym Idx ].
                Value )
218         else ( var )
219             emit LOD ( M = table [ sym Idx ].
                addr )
220         get next token
221     else if token == numbersym
222         emit LIT
223         get next token
224     else if token == lparentsym
225         get next token
226         EXPRESSION
227         if token != rparentsym
228             error
229         get next token
230     else error
231
```

Appendix E Symbol Table

This appendix provides the recommended data structure for implementing the symbol table in your compiler.

```
typedef struct
{
    int kind ;       // const = 1 , var = 2 , proc =
    char  name [ 10]; // name up to 11 chars
    int val ;        // number
    int level ;      // L level
    int addr ;       // M address
    int mark ;       // to indicate unavailable or deleted
} symbol ;

symbol  symbol_table [ MAX_SYMBOL_TABLE_SIZE  =
```

Appendix F Additional Test Cases

This appendix provides additional complex test cases to help you verify your implementation of procedures, nested procedures, and recursion.

Test Case 1: Factorial Calculation using Recursion

This program calculates the factorial of 3 (3!) using recursion:

```
var f, n;

procedure fact ;
var ans1 ;
begin
    ans1 :=n;
    n:= n -1;
    if n = 0 then f := 1 else f := 0 fi;
    if n > 0 then call fact else f := f fi;
    f:=f* ans1 ;
end
;

begin
    n :=3;
    call fact ;
    write f
end .
```

When correctly implemented, this program should compute 3! = 6 and output this value.

Test Case 2: Nested Procedures with Variable Access

This program demonstrates nested procedures with variable access across different lexical levels:

```
var x, y, z, v, w;
procedure a;
   var x, y, u,
   v; procedure
   b; var y, z,
              v;
     procedure c;
       var y, z;
       begin
         z :=1;
         x:=y+z+w
       end ;
     begin
       y:=x+u+w;
       call c
     end ;
   begin
     z :=2;
     u:=z+w;
     call b
   end ;
begin
x :=1; y :=2; z :=3; v :=4; w :=5;
   x:=v+w;
   write z;
   call a;
end .
```

This test case verifies:

- Correct implementation of nested procedures (3 levels deep)
- Proper symbol table management for variables with the same name at different lexical levels
- Correct variable access across scope boundaries
- Proper procedure calls including nested procedure calls

These test cases address key aspects of the assignment that students should verify in their implementation:

- Procedure declarations and calls
- Recursive procedure calls
- Symbol table management with variable shadowing

- Proper lexical level tracking
- Correct if-else statement handling