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

Project Overview

Over the course of the semester you will be creating a lexical analyzer, compiler, and virtual machine (VM) package for the small language PL/0. We have given you the skeleton for the package, and for each homework you will write one of the c files in package.

Package Structure
The package is controlled by "driver.c". It reads in the input file given by the user and then calls the lexical analyzer to populate the lexeme list, followed by the parser to generate code, before finally calling the virtual machine to execute the code. These three steps are each written in separate c files: "lex.c", "parser.c", and "vm.c" respectively. The three files are connected via a header file "compiler.h". The header file contains all the major function declarations so the driver can call them as well as all the package structs so programs can pass information between each other.

The driver program also reads compiler directives from the user specifying what output should be printed. These are given as command line arguments alongside the input file name at runtime. The compiler directives are as follows:

-l	the lexeme list should be printed by "lex.c"
-s	the symbol table should be printed by "parser.c"
-a	the generated assembly code should be printed by "parser.c"
-v	the program execution trace should be printed by "vm.c"

We provide you with "driver.c" and "compiler.h". It is not necessary or recommended that you change them. If you do choose to alter either of these programs, you must include clear explanations of changes and compilation instructions in your readme. We will also give you skeleton versions of "lex.c", "parser.c", and "vm.c" which contain print functions and some helper functions. You may alter these as you please without noting in the readme, but be warned that if your output doesn't match the formatting we specify, you may experience a delay in grading.

Project Instructions and Guidelines

The following guidelines will hold for all four assignments:

- Code must be written in C
- Code must run on Eustis3 (Please note that Eustis3 is a different server than Eustis) and if it does not compile on Eustis3, it will receive a grade of 0
- Submissions that do not compile will receive a grade of 0
- Students may work by themselves or with a partner. Either way, author names must be included in the submission comment, the readme, and the header comments of all submitted files
- Late submissions will be excepted for two days after the due date with a 10 point deduction for the first day, and 20 points for the second day
- Do not change the assignment specifications (ie the grammar or the ISA). This will result in a grade of 0.
- Do not cheat. This includes resubmitting an implementation from a previous semester, even your own.
- For each assignment, submit a readme.txt and the source code you implemented. For HW1, submit "vm.c". For HW2, submit "lex.c". For HW3, submit "parser.c". For HW4, submit the whole package. For the first three programs do not submit the other files in the package unless you altered them.
- If you alter the structure of the package and/or alter "driver.c" or "compiler.h", you must indicate this in your readme or you will lose 5 points.

Testing Your Work

You'll be working on one program at a time, but the package requires all three to work properly. To this end, we provide you with compiled versions of all three so you can run your package. These are the .o files. They were compiled on Eustis3, so they will only work on Eustis3. If you try to run them on your PC, you won't be able to. For HW1, compile the package with "lex.o" and "parser.o". For HW2, compile with "vm.o" and "parser.o". For HW3, compile with "vm.o" and "lex.o". You need to submit corrected and amended programs for HW4, but we use the compiled versions for the first three homeworks. We compile your programs with a make file. We include one with each homework description, but they all follow the pattern: "gcc driver.c vm.c lex.c parser.c -lm". To run with an o file instead you simply replace the .c with .o on that file.

For grading, we use a bash script to compare your output with correct output. This speeds up grading. For each homework, we give you a sample bash script for the test cases. In essence, they compile your package using the make file and for each test case, they run the package on the test case and dump the output into a buffer file ("./a.out test1.txt -v > output.txt") before comparing the buffer to correct output ("diff -w -B output.txt output1.txt”). The bash script will only tell you if the output matches; it won’t tell you what the differences are. You can see the differences by running “diff -w -B <buffer> <correct>”. This ignores whitespace and newline differences. On occasion this won’t work as desired, but as long as your output matches and is correct, you won’t lose points for formatting differences (but you might experience a delay in grading).

What if I want more test cases? Good news! You can make your own. You can compile a correct package by compiling all the .o files together “gcc driver.c vm.o lex.o parser.o -lm” OR you can use “magic”. “magic” is an executable file made on Eustis3 which is all the source code used to make the .o files compiled together. You can use it the same way you use an “a.out” file. Ex. “./magic <input file> <directives>” and you can create correct output files by adding “ > <output file name>” to the end of the execution command.

HW 1 : The Virtual Machine (VM)
In this assignment, you will implement a virtual machine (VM) known as the P-machine (PM/0). The P-machine is a stack machine with two memory areas called the “stack,” which is organized as a data-stack to store variables and which grows downwards, and the “text”, which contains the instructions for the VM to execute. The machine also has a register file where arithmetic and logic instructions are executed. The PL/0 ISA has approximately 20 instructions including call and return (for subroutines), load and store variables, jump instructions, read and write, and several arithmetic and logic instructions. The details of all of these will be supplied in the instruction document for HW1.

The file you will write this program in is “vm.c”. It contains a function “execute” which you will implement. It is passed the instructions generated by the compiler in the form of an instruction struct. Your program must load the instructions into the stack before execution. Your program will print out each line of execution as it moves through instructions so the user can trace through the program. The user can specify that this trace should be printed by using the “-v” compiler directive. The execute function is passed a flag to indicate if that directive has been used. The print statements for read and write instructions should always be printed, regardless of directives. This program is effectively last in the order of execution of the package.

HW 2 : The Lexical Analyzer
In this assignment, you will implement a lexical analyzer for PL/0. There are four kinds of tokens in this small language: numbers, identifiers, special symbols, and reserved words. Your program should also ignore spaces and comments and recognize certain lexical errors. This program shouldn’t check for grammar errors; these are found by the parser. This means that if the lexical analyzer is given a program which isn’t grammatically correct, but

is lexically correct it should return the lexeme list without issue. More details on this will be supplied in the instruction document for HW2.

The file you will write this program in is “lex.c”. It contains a function “lexanalyzer” which you must implement. It is passed a string containing the contents of the input file and should return an array of lexemes (a package struct). The user can specify that the lexeme list should be printed by using the “-l” compiler directive. This program is effectively first in the order of execution of the package.

HW 3 : The Parser/Compiler
In this assignment, you will implement a recursive descent parser and intermediate code generator for the small language PL/0. You can either implement these as two separate programs or as one. The pseudocode given in the instruction document for HW3 has them combined, and you will need to provide a readme specifying structural alterations if you choose to implement them separately. The parser is a program that reads through the lexeme list produced by the lexical analyzer and checks that the program is grammatically correct according to the grammar for PL/0. It reports errors and stops the package. We provide you will a specific error list in the HW3 document. The parser also produces the symbol table for the program. The user can specify that this should be printed with the “-s” compiler directive. The code generator uses the lexeme list and the symbol table to generate instructions. The user can specify that the code should be printed with the “-a” directive.

The file you will write this program in is “parser.c”. It contains a function “parse” which you must implement. It is passed an array of lexemes (package struct) from the lexical analyzer as well as two print flags indicating if the symbol table and/or generated code should be printed. It should return the generated code. This program is effectively middle in the order of execution of the package.

HW 4 : Language Alteration
In this assignment, you will take the three programs you have already written and implement an alteration to PL/0. This will require alterations to all three programs and “compiler.h” (the last we’ll provide for you). It is important that you correct the previous assignments before implementing the changes.

<u>Grammar (EBNF) of tiny PL/0:</u>

program ::= block "**.**" **.**
block ::= var-declaration procedure-declaration statement**.**
var-declaration ::= [“**var**" ident [“**[**“ number “**]**”] {“**,**” ident} “**;**"]**.**
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.