CS 3323: Principles of Programming Languages

Fall 2022

Dr. Richard M. Veras

Lab: Building a Compiler from Scratch

Goal: This lab is intended to expose you to the low-level details of using compiler-compiler tools like flex and bison, and to provide you with an opportunity to relate what we have covered in class to practical experience.

Due Dates:

Part 1 (Lexing):

- Regular: Due by 11:59PM on 11/23/2022
- Bonus +10 pts: Due before 11:59PM 11/21/2022
- Penalty –10pts/day, max penalty –50: After 11:59PM 11/21/2022

Part 2 (Parsing):

- Regular: Due by 11:59PM on 12/5/2022
- Bonus +10 pts: Due before 11:59PM 12/2/2022
- Penalty –10pts/day, max –50: After 11:59PM 12/5

Part 3 (Interpreting):

- Regular: Due by 11:59PM on 12/7/2022
- Bonus +10 pts: Due before 11:59PM 12/6/2022
- Penalty –10pts/day, max –20: After 11:59PM 12/7

Grading: Note that there are far in excess of 100 points possible for this lab. For each part, the first pass of evaluation will be performed by a second team that you will be peered with (see canvas). The actual grade will be determined by the Instructor and TA. The deadline for peer grading is 2 days after the regular due date or 2 days after the part was submitted, whichever comes second.

Point Allocation for each part

- Part 1: Baseline 40pts (Max with bonus 50) + 10 pts for evaluating peer team by deadline
- Part 2: Baseline 40pts (Max with bonus 50) + 10 pts for evaluating peer team by deadline
- Part 3: Baseline 40pts (Max with bonus 50) + 10 pts for evaluating peer team by deadline

Total possible group points with bonuses: 180 (technically 200, after you read the next part)

As a team you will determine the individual distribution of points:

	Member A	Member B	Member C
Percent Contribution (%)			

If as a team, you reach a consensus of the distribution your team will get an additional 20 pts. If a consensus cannot be reached, then each team member will submit what they feel was the distribution and we will take an average of the three distributions. Your final score for the lab will be:

(Member's %distribution) *3*(Total number of points for the team)

It is worth noting that you do not need to complete all three parts to earn an 'A' on this lab.

<u>Repository Structure:</u>

For all three parts you will create a gitlab (not github) repository and add myself and our TA Ega as maintainers on the project, along with your teammates. The name of the repository will be of the form: "CS3323_FA22_LAB01_TEAM_##" without quotes and the "##" replaced with your team name. The repository will have the following directory structure:

CS3323_FA22_LAB01_TEAM_##/Readme.md

CS3323_FA22_LAB01_TEAM_##/part_i/

CS3323_FA22_LAB01_TEAM_##/part_ii/

CS3323_FA22_LAB01_TEAM_##/part_iii/

Your **readme.md** should contain a **statement of work** of how the work will be divided (every member is expected to contribute substantially to each part, submissions where each member worked on a separate part will not be considered). Additionally, a **schedule** of when the members will work on what part and when the team will synchronize will be maintained throughout the duration of the project. Finally, meeting minutes will be kept throughout the duration of the lab.

All three of these need to be in the readme and will be checked by your peer teams and evaluated. For example, these artifacts started early and updated often will carry more points than those cobbled at the last minute.

Each “part” sub-directory should contain the necessary instructions (readme.md), code, makefile, testcases, and other artifacts for your project to be successfully evaluated by your peer team and by the instructors of the course.

Part I: Constructing a Lexer

Download the tarball “**cs3323_fa22_lab01.tar.gz**” from canvas and un-tar on a “**gpel**” machine in the “./part_i” directory. You should have the file “pl-scanner.yy”, “driver.cc”, “tokens.h” and “Makefile” in your directory. Then, perform the necessary modifications to the rule’s file to recognize the strings and tokens below. The token codes can be found in the “**tokens.h**” header file.

1. Add the necessary rules to recognize the arithmetic operators: +, −, ∗, /, + =, ++, <= , >=, ==, ~=, <, >. See the file tokens.h to determine the constants to be used (they start with the prefix ’OP’).
2. Modify the rule returning the T_ID token to recognize all identifiers matching the following:
    a. identifiers **must start** with the character ’@’.
    b. the character immediately following the ’@’ **must be a letter**.
    c. letters used in identifiers can only be in lower case, that is, letters ’a’ through ’z’ are valid, while ’A’ through ’Z’ are not.
    d. characters following the initial ’@’ and the first letter **can be** any letter, digit or the underscore ’_’ symbol.

    Examples of valid identifiers are (commas used separate identifiers): @iter, @iii, @a1i, @a_xe, @_ p, @b20, @a_b_c_2 .

    Some examples of invalid identifiers are (commas used separate identifiers): _, _2, at, 2_, 12b_5, abC24, @_b43, _delta, @, @Ijk.
3. Create a new rule to recognize floating point numbers. The rule should return the token L_FLOAT:
    a. Can start with a digit, the ’+’ or a ’-’
    b. Both the integer and fractional part should consist of at least one digit
    c. The integer and fractional part should be separated by a ’.’
    Examples of acceptable floating point numbers are: ”10.0”, ”+1.5”, ”-10.50000”, ”0.9”.
    Examples of strings that should be rejected are: ”0.”, ”.01”
    d. Create a rule to recognize the following keywords: INTEGER, FLOAT, FOREACH, BEGIN, END, REPEAT, UNTIL, WHILE, DECLARE, IF, THEN, PRINT. The tokens corresponding to the keywords are those with the prefix ’K ’ in the tokens.h header file.

For convenience, a **Makefile** is provided, but you are not required to use it. Run:

make

to rebuild the scanner generator (lex.yy.c), and to recompile the driver.

To validate your scanner's functionality, prepare a file containing a set of valid and invalid inputs. It is recommended to test the scanners with different groups of inputs. For example, test your identifier rule first, then integer numbers, then floating point numbers, then keywords and so on. You can redirect any of the input test file by doing:

./pl-scanner.exe < input.txt

Your grade will be determined by comparing the output produced by your scanner against our own set of inputs.

A tarball containing your Part I directory will be submitted to canvas.

The rubric for this part is as follows. As a team you will fill this out, your peer team will also fill this, as will the instructors. The final grade will be entirely determined by the instructors.

Grader	Readme (sow,sch,min) 4pts	Test cases 6pts	P1 6 pts	P2 8pts	P3 10pts	P4 6pts
Team						
Peer						
Instructors						

<u>Part II:</u> Building a parser

The objective of this part is to build a fully functional parser. Download the tarball "**cs3323_fa22_lab01.tar.gz**" from canvas and un-tar on a "**gpel**" machine in the "./part_ii" directory. You should have the files grammar.y, scanner.yy, inputs.tar.gz (decompresses to directory inputs), Makefile and driver.c. Then, perform the necessary modifications to the grammar's file to accept/reject the example programs provided. You should only work on the grammar file.

You must complete the production rules of grammar.y. While a description of each rule is provided below, you should also use the input files provided to further understand the syntactic structure that is being requested. In some cases, partial implementations of the requested productions are also given, but you will not be able to test them until the grammar is completed. Please note that the Line numbers below are relative to the unmodified grammar.y file. Once you start adding your own rules, the lines below will shift.

1. Complete the production corresponding to the **varlist** non-terminal (Lines 143–145 in grammar.y), which is used in the production of the non-terminal **read** (Line 139 in grammar.y). **varlist** should produce a comma-separated list of variable references (**varref**). The list of variable references should be of at least length one.
2. Complete the production corresponding to the **expr_list** non-terminal (Lines 147–149 in grammar.y). It should produce a comma-separated list of arithmetic expressions (See non-

terminal **a_expr**, Lines 95–98). The list of arithmetic expressions should be of at least length one.

3. Define three productions for the non-terminal **l_fact** (Lines 124–126 in grammar.y):
    a. a left-recursive rule producing comparisons of arithmetic expressions (a expr non-terminal). It should use the **oprel** non-terminal already defined.
    b. a single arithmetic expression.
    c. A logical expression in parenthesis (**l_expr** non-terminal).
4. Define two productions for the **varref** non-terminal (Line 112 in grammar.y) that match the below description:
    a. A variable reference can be the T_ID token.
    b. A variable reference can be a left-recursive list of arithmetic expressions delimited by '[' and ']'. The recursion terminates with the T_ID token (See above description).
5. Define five productions for the non-terminal **a_fact** (Lines 105–109 in grammar.y) based on the following description:
    a. An **a_fact** can be a variable reference (non-terminal **varref**).
    b. The token T_NUM.
    c. A literal string (token T_LITERAL_STR).
    d. The non-terminal **a_fact** preceded by the T SUB token (Note: Do not use '-').
6. Complete the control-flow constructs (Lines 74–91 in grammar.y). Observe that a statement list surrounded by the tokens T BEGIN and T END is also a statement. The non-terminal **l_expr** must be used for representing logical expressions. Use test cases for*.smp, if*.smp, repeat*.smp and for*.smp. The provided test cases have a suffix "pass" or "fail", right before the extension ".smp". The suffix denotes the result you should obtain from running the input file.
    a. **foreach**: Complete the partially-defined production. See input cases for[1-4] pass.smp.
    b. **repeat-until**: Define it as a list of statements. Use the non-terminal **stmt_list**). The list must be delimited by the tokens T REPEAT and T UNTIL. The controlling condition should use the **l_expr** non-terminal. Do not add parentheses. See input cases repeat*.smp.
    c. **while**: The T WHILE token followed by a logical expression and any statement. See input cases while*.smp.

A tarball containing your Part II directory will be submitted to canvas.

The rubric for this part is as follows. As a team you will fill this out, your peer team will also fill this, as will the instructors. The final grade will be entirely determined by the instructors.

Grader	Readme 4pts	Test cases6 pts	P1 2 pts	P2 2pts	P3 6pts	P4 4pts	P5 8pts	P6 8pts
Team								
Peer								
Instructors								

Part III: Finishing the compiler

The objective of this programming assignment is to build a functional compiler with basic I/O support and arithmetic computations. Instructions are provided below.

First, download the tarball "**cs3323_fa22_lab01.tar.gz**" from canvas and un-tar on a "**gpel**" machine. In the part III directory, there should be the files grammar.y, scanner.yy, icode.cc, icode.hh, symtab.cc, symtab.hh, inputs-int.tar.gz, inputs-float.tar.gz, Makefile, run-all.sh and driver.cc. Then, perform the necessary modifications to **grammar.y** and to **icode.cc** to add support for floating point operations. You should only work on the grammar file (**grammar.y**) and in **icode.cc**.

1. In grammar.y, complete the semantic actions corresponding to the floating point arithmetic binary operations of the non-terminals a expr and a term: (OP_FADD, OP_FSUB, OP_FMUL and OP_FDIV). You should query the datatype field of a temporary symbol to determine the correct operation to be generated.
2. In icode.cc, function run, implement the run-time actions for the floating point arithmetic binary operations OP_FADD, OP_FSUB, OP_FMUL and OP_FDIV. These four operations are specific to the datatype DTYPE_FLOAT.
3. In icode.cc, function run, complete the run-time action of the floating point arithmetic unary operation OP_UMIN for the DTYPE_FLOAT datatype (float). The current implementation only supports DTYPE_INT (a C/C++ int).
4. In icode.cc, function run, complete the run-time action of the operation OP_STORE for the datatype DTYPE_FLOAT. It should be very similar to the DTYPE_INT case.
5. In icode.cc, function run, complete the run-time action of the operation OP_LOAD for the datatype DTYPE_FLOAT. It should be very similar to the DTYPE_INT case.
6. In icode.cc, function run, complete the run-time action of the operation OP_LOAD_CST for the datatype DTYPE_FLOAT. It should be very similar to the DTYPE_INT case.

A number of test cases are provided, both for the int (directory inputs-int) and float (directory inputs-float) language datatypes. The current implementation of the provided compiler works with all the test cases of the directory inputs-int. You can test your progress with the files in inputs-float. Note that the output should be practically identical to the corresponding int test case.

For convenience, a Makefile is also provided, but you are not required to use it. The Makefile will build two binaries, simple.exe and simple-debug.exe. The latter will output the symbol table, instruction table and a number of debug prints. Grading will be performed with simple.exe . If you need to add debug printing information, always enclose it between "#ifdef SMP DEBUG " and "#endif".

To test a single input file, run: ./simple.exe < inputfile.smp

You can also test all the test cases of a single directory with the script run-all.sh. It expects the directory name to test.

A tarball containing your Part III directory will be submitted to canvas.

The rubric for this part is as follows. As a team you will fill this out, your peer team will also fill this, as will the instructors. The final grade will be entirely determined by the instructors.

Grader	Readme 4pts	Test cases 6pts	P1 10 pts	P2 4pts	P3 4pts	P4 4pts	P5 4pts	P6 4pts
Team								
Peer								
Instructors								

<u>Overall:</u> Part I, II, III are a max of 40pts, the bonuses are a max of 10, and peer grading is a max of 10.

Grader	Part I	Part I Bonus	Part II	Part II Bonus	Part II	Part III Bonus	Peer Grade I	Peer Grade II	Peer Grade III
Team									
Instructors									

<u>Additional Resources:</u>

…

- https://www.cs.virginia.edu/~cr4bd/flex-manual/
- http://alumni.cs.ucr.edu/~lgao/teaching/flex.html
- http://web.mit.edu/gnu/doc/html/flex_1.html
- https://westes.github.io/flex/manual/
- https://www.gnu.org/software/bison/manual
- https://www.lysator.liu.se/c/ANSI-C-grammar-y.html#multiplicative-expression

Final words: If you see any typos or errors, please do not hesitate to reach out to me or the TA.

Acknowledgements: This lab is an adaptation of several of Dr. Martin Kong's labs.