CS 314 Principles of Programming Languages
Project 1: A Compiler for the TinyL Language

THIS IS NOT A GROUP PROJECT. You may talk about the project and possible solutions in general terms, but must not share code. In this project, you will be asked to write a recursive descent LL(1) parser and code generator for the tinyL language. Your compiler will generate RISC machine instructions. To test your generated programs, you can use a virtual machine that can "run" your RISC code programs.

This document is not a complete specification of the project. You will encounter important design and implementation issues that need to be addressed in your project solution. Identifying these issues is part of the project. As a result, you need to start early, allowing time for possible revisions of your solution.

1 Background

1.1 The tinyL language

tinyL is a simple expression language that allows assignments and basic I/O operations.

<program>	::=	<stmt_list> !
<stmt_list>	::=	<stmt> <morestmts>
<morestmts>	::=	; <stmt_list> | ϵ
<stmt>	::=	<assign> | <read> | <print>
<assign>	::=	<variable> = <expr>
<read>	::=	? <variable>
<print>	::=	% <variable>
<expr>	::=	+ <expr> <expr> |
		− <expr> <expr> |
		∗ <expr> <expr> |
		& <expr> <expr> |
		| <expr> <expr> |
		<variable> |
		<digit>
<variable>	::=	a | b | c | d | e | f
<digit>	::=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

The & operator is the bitwise AND operator and the | is the bitwise OR operator. Here are two examples of valid **tinyL** programs:

```
?a;a=+2+25;%a!
?a;a=|2&3|25;%a!
```

1.2 Target Architecture

The target architecture is a simple RISC machine with virtual registers, i.e., with an unbounded number of registers. All registers can only store integer values. A RISC architecture is a load/store architecture where arithmetic instructions operate on registers rather than memory operands (memory addresses). This means that for each access to a memory location, a `load` or `store` instruction has to be generated. Here is the machine instruction set of our RISC target architecture. R_x , R_y , and R_z represent three arbitrary, but distinct registers.

instr. format	description	semantics
memory instructions		
`LOADI` R_x `#<const>`	load constant value #<const> into register R_x	$R_x \leftarrow$ <const>
`LOAD` R_x `<id>`	load value of variable <id> into register R_x	$R_x \leftarrow$ <id>
`STORE <id>` R_x	store value of register R_x into variable <id>	<id> $\leftarrow R_x$
arithmetic instructions		
`ADD` R_x R_y R_z	add contents of registers R_y and R_z , and store result into register R_x	$R_x \leftarrow R_y + R_z$
`SUB` R_x R_y R_z	subtract contents of register R_z from register R_y , and store result into register R_x	$R_x \leftarrow R_y - R_z$
`MUL` R_x R_y R_z	multiply contents of registers R_y and R_z , and store result into register R_x	$R_x \leftarrow R_y * R_z$
`AND` R_x R_y R_z	Bitwise AND contents of registers R_y and R_z , and store result into register R_x	$R_x \leftarrow R_y$ & R_z
`OR` R_x R_y R_z	Bitwise OR contents of registers R_y and R_z , and store result into register R_x	$R_x \leftarrow R_y \mid R_z$
I/O instructions		
`READ <id>`	read value of variable <id> from standard input	read(<id>)
`WRITE <id>`	write value of variable <id> to standard output	print(<id>)

2 Project Description

You can use C/C++, Python, or Java to implement the project. If you choose to use C/C++, a start package is provided, and you will only need to complete the partially implemented recursive descent LL(1) parser that generates RISC machine instructions. If you choose to use Python or Java, you must start from scratch. However, note that the simulator and the test cases for running the assembly are also in the start package. You will need to download the start package even if you do not plan to implement it in C/C++.

The project represents a programming environment consisting of a compiler and a virtual machine (RISC machine interpreter).

2.1 Compiler

The recursive descent LL(1) parser implements a simple code generator. In the C/C++ start package, you should follow the code's main structure as given to you in file `Compiler.c`. As given to you, the file contains code for function `digit` and partial code for function `expr`. The compiler can generate code only for expressions containing "+" operations and constants. You will need to add code in the provided stubs to generate the correct RISC machine code for the entire program. Do not change the signatures of the recursive functions. Note: The left-hand and right-hand occurrences of variables are treated differently. You can type "make" and it will generate a "compile" executable in the C/C++ version.

For the Python version, you should have your main function in the file named "Compiler.py". Please use Python3 (the same environment and Python configuration) for Python implementation. Your code should be compilable and executable on ilab. For the Java version, you should have the main function in the file with the name "Compiler.java". The autograder will automatically search for one of the three main files. Again, your code should be compilable and executable on ilab. In Gradescope, we will set up an environment similar to that on the ilab machines.

2.2 Virtual Machine

The virtual machine executes a RISC machine program. If a `READ <id>` instruction is executed, the user is asked for the value of `<id>` from standard input (stdin). If a `WRITE <id>` instruction is executed, the current value of `<id>` is written to standard output (stdout). The virtual machine is implemented in file `Interpreter.c`. DO NOT MODIFY this file. It is there only for your convenience so that you may be able to copy the source code of the virtual machine, for instance, to your laptop and compile it there. This is for your convenience since the project will be graded in the same environment as the ilab cluster. Type "Make run" to get the virtual machine.

The virtual machine assumes that an arbitrary number of registers are available (called virtual registers), and that there are only **6 memory** locations that can be accessed using variable names ('a' ... 'f'). In a real compiler, an additional optimization pass maps virtual registers to a machine's limited number of physical registers. This step is typically called *register allocation*. You do not have to implement register allocation in this project. The virtual machine (RISC machine language interpreter) will report the overall number of executed instructions for a given input program.

3 Grading

3.1 Basic score

For C/C++ implementation, you will submit your version of the file and `Compiler.c`. No other file should be modified, and no additional file(s) may be used. For Java implementation,

you will submit your version of `Compiler.java`, and other functions if necessary. For Python implementation, you need to submit `Compiler.py`.

Your programs will be graded based mainly on functionality. Functionality will be verified through automatic testing on syntactically correct test cases. No error handling is required. The original project distribution contains a few test cases. Note that we will use hidden test cases during grading.

A simple Makefile is also provided in the distribution for your convenience. In the C/C++ version, in order to create the compiler, say `make compile` at the Linux prompt, which will generate the executable compile.

In the C/C++ version, the provided initial compiler is able to parse and generate code for very simple programs consisting of only a single statement with a "expr". Your implementation will need to be able to accept and compile the full tinyL language.

The Makefile also contains rules to create the virtual machine (`make run`).

3.2 Extra Credit for Optimization

You can use the algorithm we discussed in class to implement the code generator. You will get full credit for each test case if your generated ILOC code runs successfully. However, you can also use various optimization techniques to improve the instruction count in this project. We will use a leaderboard to keep track of the top 9 performers that produce a minimal number of instructions for a selected set of benchmarks, including hidden test cases. The top 3 performers will get 0.5% extra credit for the entire course, the next 3 top performers will get 0.25% extra credit, and the top 7-9 performers will get 0.125% extra credit. '

Below are a few ways to optimize the number of instructions in the generated ILOC code. The examples are shown in Figure 1, where the A version is the original code, and the B version will imply a different number of ILOC instructions if your compiler considers the three types of optimizations. Note that you do not have to generate the B version of the code directly. It is just used for describing the different types of optimizations that can be performed.

- **Constant propagation**: The goal of constant propagation is to discover values that are constant on all possible executions of a program and to propagate these constant values as far forward through the program as possible.

- **Common sub-expression elimination (CSE)**: It searches for instances of identical expressions (i.e., they all evaluate to the same value), and analyzes whether it is worthwhile replacing them with a single variable holding the computed value.

- **Dead code elimination**: This technique removes code that is unused and does not affect the program's output, known as "dead code". Dead code can be entire functions that are never called, or code after a return statement that is never executed.

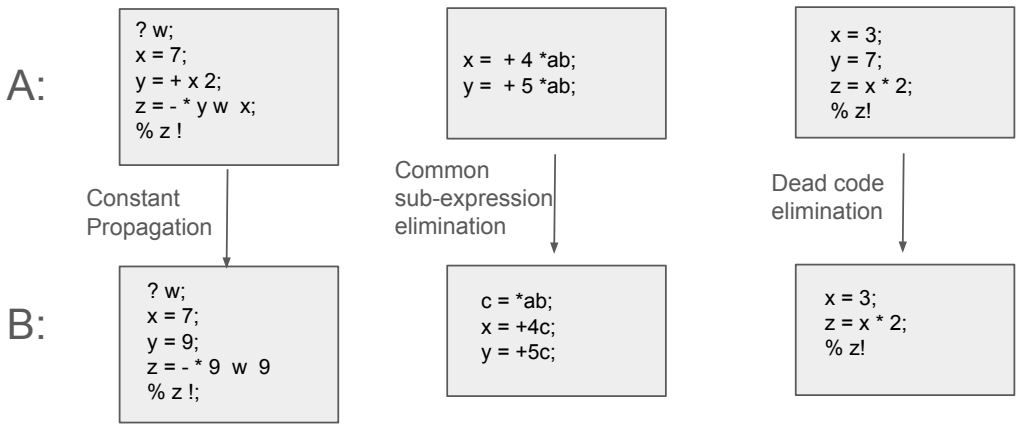

Figure 1: Examples of Optimization

4 Get Started

Download the code package from Canvas. If you want to work on it in the ilab cluster, create your directory on the ilab cluster, and copy the entire provided project folder to your home directory. Make sure that the read, write, and execute permissions for groups and others are disabled (`chmod go-rwx <directory_name>`).

Say `make compile` to generate the compiler for the C/C++ version. To run the compiler on a test case "demo.tinyL", say `./compile demo.tinyL`. This will generate a RISC machine program in file `tinyL.out`. For Python, use `python3 Compiler.py demo.tinyL`. For Java version, use `javac Compiler.java; java Compiler demo.tinyL` .

The RISC virtual machine (RISC machine program interpreter) can be generated by saying `make run`. The distributed version of the VM in `Interpreter.c` is complete and should not be changed. To run a program on the virtual machine, for instance `tinyL.out`, say `./run tinyL.out`. If the program contains `READ` instructions, you will be prompted at the Linux command line to enter a value. Finally, you can define a **tinyL language interpreter** on a single Linux command line as follows:

`./compile demo.tinyL; ./run tinyL.out`

Please note that when using Python or Java, the format for printed output, when the command is `WRITE`, should adhere to the format used in the C version provided in the start package. Specifically, the output should be: `tinyL>> [VARIABLE NAME] = [VALUE]`. For example: `tinyL>> a = 6`. In summary, ensure that all output formats are consistent across different languages as established in the C version.

The test cases are included in the "tests" folder. There are 15 test cases given and 5 test cases hidden. We will use all 20 test cases to evaluate your code implementation. Your code will strictly be graded based on functionality.

5 Questions

All questions regarding this project should be posted on the Piazza site. Good luck!