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. You will also write a code optimizer that takes RISC machine instructions as input and removes redundant code. The output of the optimizer is a sequence of RISC machine instructions which produces the same results as the original input sequence but is more efficient. To test your generated programs, you will use a provided virtual machine that can "run" your RISC code. The project will require you to manipulate doubly-linked lists of instructions. In order to avoid memory leaks, explicit deallocation of unused memory space is necessary.

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>	::=	<var> = <expr>
<read>	::=	% <var>
<print>	::=	$ <var>
<expr>	::=	<arith_expr> |
		<logical_expr> |
		<var> |
		<digit>
<arith_expr>	::=	+ <expr> <expr> |
		− <expr> <expr> |
		∗ <expr> <expr> |
<logical_expr>	::=	& <expr> <expr> |
		| <expr> <expr> |
<var>	::=	a | b | c | d | e | f
<digit>	::=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Two examples of valid **tinyL** programs:

1. `%a;%b;c=&3*ab;d=+c1;$d!`

2. `%a;b=-*+1+2a58;$b!`

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	$R_x \leftarrow R_y * R_z$
logical instructions		
`AND` R_x R_y R_z	apply bit wise AND operation to 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	apply bit wise OR operation to 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>)

1.3 Dead Code Elimination

Our tinyL language does not contain any control flow constructs (e.g.: jumps, if-then-else, while). This means that every generated instruction will be executed. However, if the execution of an operation or instruction does not contribute to the input/output behavior of the program, the instruction is considered "dead code" and therefore can be eliminated without changing the semantics of the program. Please note that `READ` instructions may not be eliminated although the read value may have no impact on the outcome of the program.

All `READ` instructions are kept since the overall input/output program behavior needs to be preserved.

Your dead-code eliminator will take a list of RISC instructions as input. For example, in the following code

```
LOADI Rx #c1
LOADI Ry #c2
LOADI Rz #c3
ADD R1 Rx Ry
MUL R2 Rx Ry
STORE a R1
WRITE a
```

the `MUL` instruction and the third `LOADI` instruction can be deleted without changing the semantics of the program. Therefore, your dead-eliminator should produce the code:

```
LOADI Rx #c1
LOADI Ry #c2
ADD R1 Rx Ry
STORE a R1
WRITE a
```

2 Project Description

The project consists of two components:

1. Complete the partially implemented recursive descent LL(1) parser that generates RISC machine instructions.
2. Write a dead-code eliminator that recognizes and deletes redundant, i.e., dead RISC machine instructions.

The project represents an entire programming environment consisting of a compiler, an optimizer, and a virtual machine (RISC machine interpreter). You are required to implement the compiler and the optimizer (described in Section 2.1 and 2.2). The virtual machine is provided to you (described in Section 2.3).

2.1 Compiler

The recursive descent LL(1) parser implements a simple code generator. You should follow the main structure of the code as given to you in file `Compiler.c`. As given to you, the file

contains code for function `digit`, `assign`, and `print`, as well as partial code for function `expr`. As is, the compiler is able to generate code only for expressions that contain "+" operations and constants. You will need to add code in the provided stubs to generate 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.

2.2 Dead Code Elimination Optimization

The dead code elimination optimizer expect the input file to be provided at the standard input (stdin), and will write the generated code back to standard output (stdout).

The basic algorithm identifies "crucial" instructions. The initial crucial instructions are all `READ` and `WRITE` instructions. For all `WRITE` instruction, the algorithm has to detect all instructions that contribute to the value of the variable that is written out. First, you will need to find the `store` instruction that stores a value into the variable that is written out. This `STORE` instruction is marked as critical and will reference a register, let's say *Rm*. Then you will find the instructions on which the computation of the register *Rm* depends on and mark them as critical. This marking process terminates when no more instructions can be marked as critical. If this basic algorithm is performed for all `WRITE` instructions, the instructions that were not marked critical can be deleted.

Instructions that are deleted as part of the optimization process have to be explicitly deallocated using the C `free` command in order to avoid memory leaks. You will implement your *dead code elimination* pass in the file `Optimizer.c`. All of your "helper" functions should be implemented in this file.

2.3 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. Be aware that you are welcome to work on your project using your own computer, however, you need to make sure your code will eventually compile and run on the ilab cluster. All grading will be done on ilab.

The virtual machine assumes that an arbitrary number of registers are available (called virtual registers), and that there are only six memory locations that can be accessed using variable names ('a' ... 'e' 'f'). In a real compiler, an additional optimization pass maps virtual registers to the limited number of physical registers of a machine. 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. This allows you to assess the effectiveness of your optimization component.

3 Grading

You will submit two files `Optimizer.c` and `Compiler.c`. No other file should be modified, and no additional file(s) may be used. Please make a tarball of these two files: "tar -cvf proj1_submission.tar Optimizer.c Compiler.c" and submit the *proj1_submission.tar* file. Do not submit any executables or any of your test cases.

Your programs will be graded based mainly on functionality. Functionality will be verified through automatic testing on a set of syntactically correct test cases. No error handing is required. The project code package contains 10 test cases. Note that during grading we will use another 10 hidden test cases. Your grade will be based on these 20 test cases.

The project code package also contains executables of reference solutions for the compiler (compile.sol) and optimizer (optimize.sol).

A simple Makefile is provided in the package for your convenience. In order to create the compiler, type `make compile` at the Linux prompt, which will generate the executable compile. The Makefile contains rules to create executables of your optimizer (`make optimize`) and virtual machine (`make run`). The Makefile also contains a rule that cleans all the object files before you recompile your code.

The provided, initial compiler contains some code for parsing a single assignment statement with right-hand side expressions of only additions of numbers, followed by a single print statement. The provided code is by no means complete (it is only meant to show you the code framework). You will need to extend it and support the complete *TinyL* language.

4 How To Get Started

Create your own directory on the ilab cluster, and copy the provided project code package to your directory. Make sure that the read, write, and execute permissions for groups and others are disabled (`chmod go-rwx <directory_name>`).

Type `make compile` to generate the compiler. To run the compiler on a test case "sample2.tinyL" in the folder tests/, type `./compile tests/sample2.tinyL`. This will generate a RISC machine program in the file `tinyL.out`. To create your optimizer, type `make optimize`. The initial (provided) version of the optimizer does not work at all.

To call your optimizer on a file that contains RISC machine code, for instance file `tinyL.out`, type `./optimize < tinyL.out > optimized.out`. This will generate a new file `optimized.out` containing the output of your optimizer. The operators "<" and ">" are Linux redirection operators for standard input (stdin) and standard output (stdout), respectively. Without those, the optimizer will expect instructions to be entered on the Linux command line, and will write the output to your screen.

You may want to use valgrind for memory leak detection. We recommend to use the following flags, in this case to test the optimizer for memory leaks:

`valgrind --leak-check=full --show-reachable=yes --track-origins=yes ./optimize < tinyL.out`

The RISC virtual machine (RISC machine program interpreter) can be generated by typing `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`, type `./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 test1; ./optimize < tinyL.out > opt.out; ./run opt.out`.

The ";" operator allows you to specify a sequence of Linux commands on a single command line.

5 Questions

Please post all questions regarding the project on Sakai forum.