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 implements constant propagation elimination. The output of the optimzer is a sequence of RISC machine instructions which produces the same results as the original input sequence. To test your generated programs, you can use a virtual machine that can "run" your RISC code programs. The project will require you to manipulate doubly-linked lists of instructions. In order to avoid memory leaks, explicit deallocation of removed instructions is necessary.

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> |
		<variable> |
		<digit>
<variable>	::=	a | b | c | d | e
<digit>	::=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Examples of valid **tinyL** programs:

```
!a;!b;c=+3*ab;d=+c1;#d.
!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	$R_x \leftarrow R_y * R_z$
	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 Constant propagation

Your optimizer will implement multiple passes of peephole optimization until no more constant propagation can be done. A constant propagation looks for a pattern of the following form:

LOADI R_x #c_1
LOADI R_y #c_2
op R_z R_x R_y

If this pattern is detected, the value of constants `c1 op c2` is computed as constant `c3`, where `op` can be addition `ADD`, subtraction `SUB`, or multiplication `MUL`. The original sequence of three instructions is then replaced by a single instruction of the form:

LOADI R_z #c_3

If no pattern is detected, the window is moved one instruction down the list of instructions. In the case of a successful match and code replacement, the first instruction of the new window is set to the instruction that immediately follows the three instructions of the pattern in the original, unoptimized code.

You might need more than one pass to exploit all the constant propagation opportunities. Below is an example:

LOADI R_a #1
LOADI R_b #1
ADD R_c R_a R_b
LOADI R_d #2
LOADI R_e #2
ADD R_f R_e R_d
ADD R_g R_f R_c

After one pass, it is optimized to

LOADI R_c #2
LOADI R_f #4
ADD R_g R_f R_c

It can be further optimized to

LOADI R_g #6

You may want to define one pass (one pass scans the entire program once) as one step, then repeat this step until no more instructions can be folded.

2 Project Description

The project consists of two parts:

a) Complete the partially implemented recursive descent LL(1) parser that generates RISC machine instructions.

b) Write a constant propagation optimization that use multiple passes, each pass use a sliding window of exactly 3 instructions

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

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`, 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 Peephole Optimizer for Constant Propagation

The peephole optimizer expects the input file to be provided at the standard input (stdin), and will write the generated code back to standard output (stdout). 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 constant propagation optimization pass in file `Optimizer.c`.

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. Note that this is for your convenience only since the project will be graded on the ilab cluster.

The virtual machine assumes that an arbitrary number of registers are available (called virtual registers), and that there are only five memory locations that can be accessed using variable names ('a' ... 'e'). 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 constant propagation elimination optimization. You also will be able to check for correctness of your optimization pass.

3 Grading

You will submit your versions of 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 original project distribution contains some test cases. Note that during grading we will use hidden test cases. The distribution also contains executables of reference solutions for the compiler (compile.sol) and optimizer (optimize.sol). A simple Makefile is also provided in the distribution for your convenience. In order to create the compiler, say `make compile` at the Linux prompt, which will generate the executable compile.

The provided, initial compiler is able to parse and generate code for very simple programs consisting of a single assignment statement with right-hand side expresssions of only additions of numbers, followed by a single print statement. You will need to be able to accept and compile the full tinyL language.

The Makefile also contains rules to create executables of your optimizer (`make optimize`) and virtual machine (`make run`).

4 How To Get Started

The code package for you to start with is provided on Sakai.

Create your own directory on the ilab cluster, and copy the entire provided project folder to your own home directory or any other one of your directories. 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. To run the compiler on a test case "test1", say `./compile test1`. This will generate a RISC machine program in file `tinyL.out`. To create your optimizer, say `make optimize`. The distributed version of the optimizer does not work at all, and the compiler can only handle a single example program structure consisting of a aingle assignment statement followed by a print statemet. An example test case that the provided compiler can handle is given in file `tests/test-dummy`.

To call your optimizer on a file that contains RISC machine code, for instance file `tinyL.out`, say `./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 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 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

All questions regarding this project should be posted on Sakai forum. Enjoy the project!