CS 314 Principles of Programming Languages

C Programming Project

Due date: Friday, October 24, 11:59pm

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 as discussed in class. Your compiler will generate RISC machine instructions. You will also write a code optimizer that takes RISC machine instructions as input and implements peephole constant propagation. 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 implement and manipulate doubly-linked lists of instructions. In order to avoid memory leaks, explicit deallocation of constant folded 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> | <swap> | <read> | <print>
<assign>	::=	<variable> = <expr>
<swap>	::=	% <variable> <variable>
<read>	::=	& <variable>
<print>	::=	# <variable>
<expr>	::=	+ <expr> <expr> |
		− <expr> <expr> |
		∗ <expr> <expr> |
		<variable> |
		<digit>
<variable>	::=	a | b | c | d
<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>)

Note that the layout of the instructions is slightly different from the one we used in class.

1.3 Peephole Optimizer

The peephole optimizer uses a sliding window of up to three RISC machine instructions. It looks for a patterns within this window of instructions, and if found, replaces the detected instruction pattern by another pattern. 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 instructions of the pattern in the original, unoptimized code.

This optimization requires that your RISC machine code has a particular shape, i.e., follows particular **code shape conventions**. The code has to use new ("fresh") virtual registers each time a value is loaded/computed into a register. In other words, the target register of any instruction is unique. In addition, any results from a load or computation cannot be reused (no common subexpressions). For example, it is not possible to have code that keeps the value "1" in a dedicated register, let's say r_1, and then each time a constant 1 occurs in the program, the code uses r_1 instead of the instruction `LOADI` r_x `#1` and register r_x, where r_x is some fresh virtual register. The reason for this restriction is that we want to be able to delete instructions based on only looking at the three instructions in our sliding window. Without the restriction, deleting an instruction is only safe if the target register in never used again in the program, which requires instructions to be considered outside the limited peephole window. *Our basic compilation scheme as discussed in class (syntax-directed translation, compiler example) generates code that satisfies these shape requirements.*

In this project, you will implement two peephole optimizations, namely *constant folding* and *algebraic simplification*.

Constant Folding – The peephole optimizer uses a sliding window of three RISC machine instructions. It looks for a pattern of the following form:

```
LOADI Rx #c1
LOADI Ry #c2
op Rz Rx Ry   // or op Rz Ry Rx
```

If this pattern is detected, the value of constants c_1 `op` c_2 is computed as constant c_3, 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 Rz #c3
```

Algebraic Simplification – This optimization uses a sliding window of three RISC machine instructions. It looks for patterns that represent algebraic operations that can be simplified:

$(a + 0)$, $(0 + a)$, $(a - 0)$, $(a - a)$, $(0 * a)$, $(a * 0)$, $(1 * a)$, and $(a * 1)$.

For example, the code sequence

```
LOADI Rx #0
LOAD Ry <id>
MUL Rz Rx Ry
```

can be replaced by

```
LOADI Rz #0
```

and the code sequence

```
LOADI Rx #1
LOAD Ry <id>
MUL Rz Rx Ry
```

can be replaced by

```
LOAD Rz <id>
```

2 Project Description

The project consists of three parts:

1. Write a recursive descent LL(1) parser that generates RISC machine instructions that respects the code shape convention as discussed in Section 1.3.
2. Write a peephole optimizer that performs algebraic simplification followed by constant folding. Each optization uses a window of exactly three RISC machine 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` and 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.

2.2 Peephole Optimizer for Constant Propagation and Algebraic Simplification

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). This allows the specification of multiple passes of the optimizations using the UNIX pipe feature. For example, to apply optimization `optimize` three times in a row, with file "tinyL.out" as input, and file "optimized.out" as output, we would specify

```
./optimize < tinyL.out | ./optimize | ./optimize > optimized.out
```

Each invocation of the optimizer traverses the linked list of instructions twice. During the first pass, it performs algebraic simplification, which is then followed by constant folding.

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 combined algebraic simplification and 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 four memory locations that can be accessed using variable names ('a' ... 'd'). 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*. 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 peephole optimizations. 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. The electronic submission procedure will be posted later. 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 additional test cases not known to you in advance. 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 Makefile also contains rules to create executables of your optimizer (`make optimize`) and virtual machine (`make run`).

We will not test for memory leaks.

4 How To Get Started

Create your own directory on the ilab cluster, and copy the entire provided project `proj1.tar` to your own home directory or any other one of your directories. Say `tar -cf proj1.tar` to extract the project files. 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 compiler and optimizer do not work, but give some code structure that you need to follow. The compiler is able to generate code for expressions consisting of the + operator and constant operands.

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. The stream operator | allows you to build pipes where the standard output of one program is forwarded to the standard input of the second program "prog1 | prog2". This allows you to run your optimization passes multiple times.

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. Enjoy the project!