CS 314 Principles of Programming Languages

Spring 2023

A Compiler and Optimization Passes for tinyL

Due date: Friday, March 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 largely given a recursive descent LL(1) parser and code generator for the tinyL language as discussed in class. Your compiler will generate RISC machine instructions called ILOC (Intermediate Language for Optimizing Compilers). You will write several code optimization passes that takes ILOC instructions as input. Their output is a sequence of ILOC instructions which produces the same results as the original input sequence. To test your generated programs, you can use a virtual machine (simulator) that can "run" your ILOC programs. The project will require you to manipulate linked lists of instructions. In order to avoid memory leaks, explicit deallocation of "eliminated" 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.

<program>	::=	<stmt_list> .
<stmt_list>	::=	<stmt> <morestmts>
<morestmts>	::=	; <stmt_list> | ϵ
<stmt>	::=	<assign> | <print>
<assign>	::=	<variable> = <expr>
<print>	::=	! <variable>
<expr>	::=	+ <expr> <expr> |
		− <expr> <expr> |
		∗ <expr> <expr> |
		/ <expr> <expr> |
		<variable> |
		<digit>
<variable>	::=	a | b | c | d | e | f | g | h | i | j
<digit>	::=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Figure 1: The tinyL language as specified by a context-free grammar

1 Clarifications and Updates

Thursday, March 9, 2023

Updated project sample solutions and two more test cases are now availabe on Canvas: `proj1-Update-March9.tar` .

All provided sample solutions should guide you in the implementation of your project. **Your optimization passes should be at least as effective as the provided sample solutions. Memory leaks may be present in the sample solutions, but your solution should not have memory leaks.**

- Dead Code Elimination: A sample solution has been provided as executable dead-code.sol .
- Constant Folding: The provided sample solution sets the next window to the code following a window where constant folding was successfully performed. Your solution may instead slide the window "down" by only a single instruction, whether constant folding has been successfully performed or not. Both, constant folding and strength reduction must not change the number of ILOC instructions, but may change the type of instructions.
- Constant Folding: You must not remove any `loadI` instructions as part of the optimization. If instructions are "dead" after constant folding, a dead code elimination pass will remove these instructions.
- Strength Reduction: The provided sample solution uses only an instruction window of size 2. Your solution must use an instruction window of size 3.
- Strenth Reduction: You must not remove any `loadI` instructions as part of the optimization. If instructions are "dead" after constant folding, a dead code elimination pass will remove these instructions.
- Strength Reduction: An updated sample solution pass has been provided

2 Background

2.1 The tinyL language

tinyL is a simple expression language that allows assignments, and print as its only I/O operation. Memory may be initialized with values before running the program using the simulator. Every token is a **single** character of the input. This makes scanning rather easy, but does not allow integer constants of more than one digit, or variable names of more than one character. The language specification in Backus-Naur from is shown above. The following are examples of two valid **tinyL** programs:

(1) `a=3;c=/3*ab;d=+c1;!d.`
(2) `a=7;b=-*+1+2a58;!b.`

2.2 Target Architecture

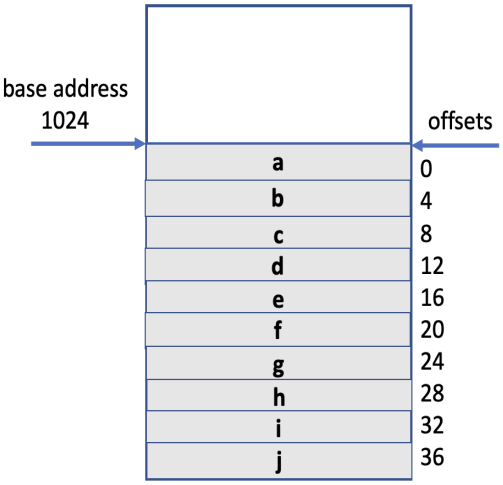
instr. format	description	semantics
memory instructions		
`loadI` $c \Rightarrow r_x$	load constant value $c$into register r_x	$r_x \leftarrow c$
`loadAI` r_x, $c \Rightarrow r_y$	load value of MEM(r_x+ c) into r_y	$r_y \leftarrow$ MEM(r_x+ c)
`storeAI` $r_x \Rightarrow r_y$, c	store value in $r_x$into MEM(r_y+ c)	MEM(r_y+ c) $\leftarrow r_x$
bit-shift operations		
`lshiftI` r_x, $c \Rightarrow r_z$	shift contents of registers $r_x$by c positions to the left store result into register r_z	$r_z \leftarrow r_x << c$
`rshiftI` r_x, $c \Rightarrow r_z$	shift contents of registers $r_x$by c positions to the right store result into register r_z	$r_z \leftarrow r_x >> c$
arithmetic instructions		
`add` r_x, $r_y \Rightarrow r_z$	add contents of registers $r_x$and r_y, and store result into register r_z	$r_z \leftarrow r_x + r_y$
`sub` r_x, $r_y \Rightarrow r_z$	subtract contents of register $r_y$from register r_x, and store result into register r_z	$r_z \leftarrow r_x - r_y$
`mult` r_x, $r_y \Rightarrow r_z$	multiply contents of registers $r_x$and r_y, and store result into register r_z	$r_z \leftarrow r_x * r_y$
`div` r_x, $r_y \Rightarrow r_z$	divide contents of registers $r_x$and r_y, and store result into register r_z	$r_z \leftarrow r_x / r_y$
I/O instruction		
`outputAI` r_x, c	write value of MEM(r_x+ c) to standard output	print(MEM(r_x+ c)

The target architecture is a RISC machine with 4096 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. You are only allowed to use these ILOC instructions. ILOC instructions are case sensitive. r_x, r_y, and $r_z$represent three registers.

2.3 Code Shape

Your compiler will generate code of a specific form with respect to how variables are accessed. All variables accesses use an address that consists of a **base pointer** and an **offset** relative to this base pointer (see picture below). The base pointer address is stored in a special register, in our case r_0. All memory references are therefore of the form MEM(r_0 + offset).

This is what the instructions `loadAI, storeAI` and `outputAI` use. All addresses are **byte** addresses. Your compiler should assign the address 1024 to r_0 at the beginning of the program. For our example language, variable offsets are non-negative byte addresses. Your compiler maps variable "a" to offset 0, variable "b" to offset 4, variable "c" to offset 8, etc. Other mappings are also possible, so we are really talking about "code shape" here, which is a particular coding style.



Your compiler generates code that does not "reuse" registers. If you assign a value to a register by a `loadI, loadAI, add, sub, mult` or `div` instruction, you will always use a fresh, i.e., new register. Your target machine has many registers, so do not worry about running out of registers. For example, this means that the code generated for `+ 1 1` will generate two `loadI` instructions with two distinct target registers and one `add` instruction. In other words, each computed value gets its own register. Your code optimization passes should handle programs that are generated by the tinyL compiler, and hand-written programs where registers may be "reused" if they contain the same value. This is called "static single assigment form", i.e., there is only a single instruction that computes the value written into a target register. For example, every occurrence of a 1 in the program is implemented by a single `loadI` at the first use of the constant with a fresh target register, and after that the target register is reused, avoiding additional `loadI` instructions. 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*. We do not deal with register allocation here.

Our tinyL language does not contain any control flow constructs (e.g.: jumps, if-then-else, while). This means that every generated instruction in the instruction sequence will be executed.

3 Project Description

The project consists of several parts:

1. Complete the partially implemented recursive descent LL(1) parser that generates ILOC instructions.
2. Write a peephole optimization pass for constant folding
3. Write a peephole optimization pass for operator strength reduction.
4. Write an optimization pass for dead code elimination

All optimization pass read ILOC instructions from standard input and write instructions back to standard output.

The project represents an entire programming environment consisting of a compiler, an optimizer, and a simulator (virtual machine) for ILOC. The ILOC simulator is called **sim** and will be made available to you as an executable on the ilab machines. This will allow you to check for correctness of your generated and optimized code.

3.1 Compiler

The recursive descent LL(1) parser implements a simple code generator. You must follow the main structure of the code as given to you in file `Compiler.c`. Do not change the signatures of the recursive functions. The code sections you need to add start with comment `/* STUDENTS - BEGIN */` and end with `/* STUDENTS - END */`. Do not modify any other code. As is, the distributed code can generate ILOC instructions for expressions with only the "+" operator (see test case "tests/bogus-program"). You will need to fix this.

Note: The left-hand and right-hand occurrences of variables are treated differently, somewhat violating our basic rules for constructing a recurse descent parser.

3.2 I/O Instruction Utility

Within an **optimization pass**, a sequence of ILOC instructions is represented as a doubly-linked list. Please see files `InstrUtils.h` and `InstrUtils.c`. Your optimization passes will work on and modify the doubly-linked list of ILOC instructions. The following two utility functions are the way your passes will read in and write out ILOC code.

```
Instruction *ReadInstructionList(FILE *infile);
```

and

```
void PrintInstructionList(FILE *outfile, Instruction *instr);
```

3.3 Optimization Passes

You will implement two peephole optimization passes and one "regular" optimization pass which needs to consider larger groups of instructions.

The two types of peephole optimizations are constant folding and operator strength reduction. Peephole optimizations look for particular instruction subsequences that allow the replaced of an instruction by a more efficient instruction or instruction sequence.

Your peephole optimization passes use a sliding window of three RISC machine instructions. It looks for code patterns as described below. 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 that have been replaced, i.e., is set to the instruction immediately after the pattern in the unoptimized code.

A peephole optimization pass (constfolding or strengthreduct) expects the ILOC 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 same optimization or different sequences of optimization passes using the UNIX pipe feature. For example, to apply optimization constfolding twice, followed by a strengthreduction pass strengthreduct with file "tinyL.out" as input, and file "optimized.out" as output, we would specify

```
./constfolding < tinyL.out | ./constfolding | ./strengthreduct > optimized.out
```

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 two peephole optimization passes in file `ConstFolding.c` and `StrengthReduction.c`.

3.3.1 Constant folding - peephole

Patterns are consecutive instructions in your ILOC program. The pattern for constant folding is a follows:

```
loadI c1 => ra
loadI c2 => rb
OP ra, rb  => rc
```

where c_1 and c_2 are integer constant, and `OP` is an arithmetic operation (`add, sub, mult`). Note: We do not apply this optimization for the division operator. The above pattern should be replaced by the code sequence

```
loadI c1 => ra
loadI c2 => rb
loadI c3 => rc
```

where c_3 is the integer constant that represents the result of the computation (c_1 OP c_2), which is done at compile time. Note that due to the code shape assumption, register values in r_a and r_b are not used after the optimized pattern, which would allow us to delete the two `loadI` instructions as part of the constant folding pass. However, to be more general and handle some hand-written code, we will leave it to the dead code deletion pass to eliminate these `loadI` instructions if indeed their target registers are not used in subsequent instructions.

3.3.2 Operator strength reduction - peephole

The patterns for this optimization are as follows:

```
loadI c1 ⇒ ra
OP rb, ra  ⇒ rc
```

where c_1 is an integer constant and a power of 2 (e.g.: 2, 4, 8, 16, ...) and `OP` is the arithmetic operation `mult` or `div`. Multiplication and division can be implemented as left-shift or right-shift operations, respectively. Note that multiplication is commutative, so there is more than one pattern due to that reason. For example,

```
loadI 4 ⇒ ra
div rb, ra  ⇒ rc
```

can be replaced by

```
loadI 4 ⇒ ra
rshiftI rb, 2 ⇒ rc
```

and

```
loadI 4 ⇒ ra
mult rb, ra  ⇒ rc
```

can be replaced by

```
loadI 4 ⇒ ra
lshiftI rb, 2 ⇒ rc
```

As in the constant folding case, do not remove the `loadI` instruction since hand-written code may reuse the register in later parts of the ILOC code. Your dead-code elimination pass should remove the instruction if it is safe to do so. The window size of 3 is used to include patterns where there is an instruction between the `loadI` instruction and the `mult` or `div` instructions.

Note: Arithmetic integer operations may result in overflow or underflow conditions due to the finite bit-width of the integer representation. However, this is not a problem for these peephole optimizations since the overflow/underflow would also occur in the unoptimized code.

3.3.3 Dead Code Elimination - regular

This is not a peephole optimization but may look at the entire or partial list of ILOC instructions.

Our tinyL language does not contain any explicit 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.

```
loadI 1024 ⇒ r0
loadI c1 ⇒ rx
loadI c2 ⇒ ry
loadI c3 ⇒ rz
add rx, ry  ⇒ rs
mult rx, ry  ⇒ rt
storeAI rs  ⇒ r0, offset
outputAI r0, offset
```

the `mult` 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 1024 ⇒ r0
loadI c1 ⇒ rx
loadI c2 ⇒ ry
add rx, ry  ⇒ rs
storeAI rs  ⇒ r0, offset
outputAI r0, offset
```

3.4 ILOC Simulator

The virtual machine executes ILOC programs. If a `outputAI` instruction is executed, the value of the specified memory location is written to standard output (stdout). This is

the only output instruction allowed for our programs. You can input values to a program through command-line parameters as discussed below. All values are of type integer. An ILOC simulator is provided as the executable (**sim**).

Due to our code shape convention (please see section on code shape above), variables 'a' through "j' are assigned fixed offsets from the base address 1024 as described in the code shape section. In order to provide an input to an ILOC program before its execution, you need to call the simulator using the **-i** option which allows you to assign values to memory locations before program "execution".

```
./sim -i 1024 3 7 < tinyL.out
```

will write the 4-byte integer values 3 and 7 to memory locations starting at address 1024. Given the ILOC code for example tinyL program `a=5;c=+ab;!c.`, the program will print 12. Note that the value of 3 in variable 'a' will be overwritten by 5 due to `a=5`.

The ILOC simulator reports the overall number of executed instructions for a given input program. This allows you (and the grader) to assess the effectiveness of your optimization pass or passes. You also will be able to check for correctness of your optimization pass.

4 Grading

You will submit your versions of files `ConstFolding.c`, `StrengthReduction.c`, `DeadCodeElimination.c` and `Compiler.c`. You may also submit an optional `ReadMe` file if you want to communicate something about your code to the grader. **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. We are planning or have made available executables of reference solutions for the compiler (compile.sol) and the optimization passes (constfolding.sol, strenghtreduct.sol, and deadcode.sol). A simple Makefile is also provided in the distribution for your convenience. For example, 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 optimization passes.

The provided, initial compiler does not work correctly since code is missing.

5 How To Get Started

The code for this project, sample solutions and test cases have been posted on our canvas site (Files/Projects).

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 -xf 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>`). **IT IS CONSIDERED CHEATING IF YOU DO NOT PROTECT YOUR PROJECT FILES**.

Say `make compile` to generate the compiler. To run the compiler on the test case "tests/test1", say `./compile tests/test1`. This will generate an ILOC program in file `tinyL.out`. To create an optimization pass, say `make constfolding`, `make strengthreduct` or `make deadcode`. By just saying `make`, the compiler and all optimization passes will be compiled. The distributed versions of the three optimization passes only read in the sequence of ILOC instruction, and then prints them out without any changes. The compiler is missing actions for some rules.

You should 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 ./constfolding
< tinyL.out
```

6 Questions

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