Assignment 1
Instruction Selection and Register Allocation
Sample Solution

15-411: Compiler Design
Rob Arnold (rdarnold@andrew) and Eugene Marinelli (emarinel@andrew)

Due: Tuesday, September 9, 2008 (1:30 pm)

Note: Instruction orders, register and temporary numberings, and interference graphs may be different but still correct depending on how the algorithms are carried out. This document serves as a correct example.

Reminder: Assignments are individual assignments, not done in pairs. The work must be all your own.

You may hand in a handwritten solution or a printout of a typeset solution at the beginning of lecture on Tuesday, September 9. Please read the late policy for written assignments on the course web page. If you decide not to typeset your answers, make sure the text and pictures are legible and clear.

In this assignment you will generate machine code to evaluate a simple arithmetic expression, carrying out some non-trivial compilation steps. The expression is

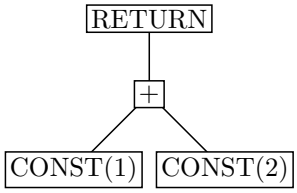
```
{ return (7 + 3 * (8 - 2)) - 4 * (9 + 5); }
```

Problem 1 (20 points)

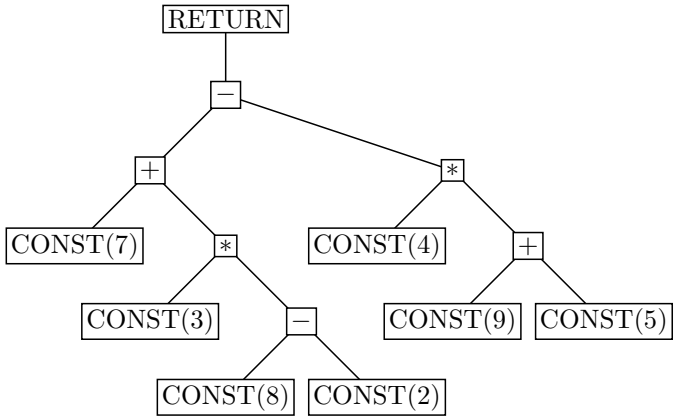
(a) Construct an abstract syntax tree (AST) corresponding to the expression. For example, the AST for the expression

```
{ return 1 + 2; }
```

would be represented as[1]:



[1]The LaTeX graph library used to generate these trees can be found at `http://www.cs.umu.se/~drewes/graphs/`. The library apparently doesn't work with `pdflatex`. You can work around this using `latex` and `dvipdf`, for example.



(b) Translate the AST from problem 1(a) into linear three-address form by applying maximal munch using the tiles in the table below. Temporaries should be called `t0`, `t1`, ..., `t`n.

Tile	IR
CONST(n)	`t`i `<-` n`;`
RETURN	`return t`i`;`
+	`t`i `<-` `t`j `+` `t`k`;`
-	`t`i `<-` `t`j `-` `t`k`;`
*	`t`i `<-` `t`j `*` `t`k`;`

For example, the AST from part (a) would be translated to

```
t0 <- 1;
```

```
t1 <- 2;
t2 <- t0 + t1;
return t2;
```

```
t0 <- 7;
t1 <- 3;
t2 <- 8;
t3 <- 2;
t4 <- t2 - t3;
t5 <- t1 * t4;
t6 <- t0 + t5;
t7 <- 4;
t8 <- 9;
t9 <- 5;
t10 <- t8 + t9;
t11 <- t7 * t10;
t12 <- t6 - t11;
return t12;
```

Problem 2 (20 points)

(a) Compute the live variables after each statement in the program generated in problem 1b.

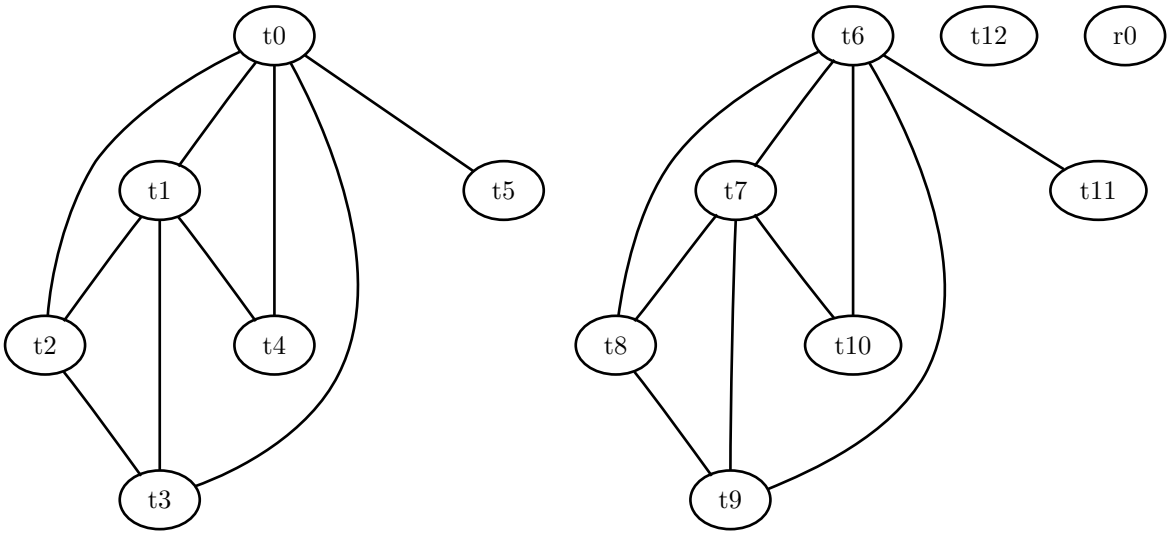
The result in returned in `r0`.

Line	Live-in temps
`t0 <- 7;`	.
`t1 <- 3;`	`t0`
`t2 <- 8;`	`t0, t1`
`t3 <- 2;`	`t0, t1, t2`
`t4 <- t2 - t3;`	`t0, t1, t2, t3`
`t5 <- t1 * t4;`	`t0, t1, t4`
`t6 <- t0 + t5;`	`t0, t5`
`t7 <- 4;`	`t6`
`t8 <- 9;`	`t6, t7`
`t9 <- 5;`	`t6, t7, t8`
`t10 <- t8 + t9;`	`t6, t7, t8, t9`
`t11 <- t7 * t10;`	`t6, t7, t10`
`t12 <- t6 - t11;`	`t6, t11`
`return t12;`	`t12`
	`r0`

(b) Construct the interference graph. Is it chordal[2]?

The graph is chordal.

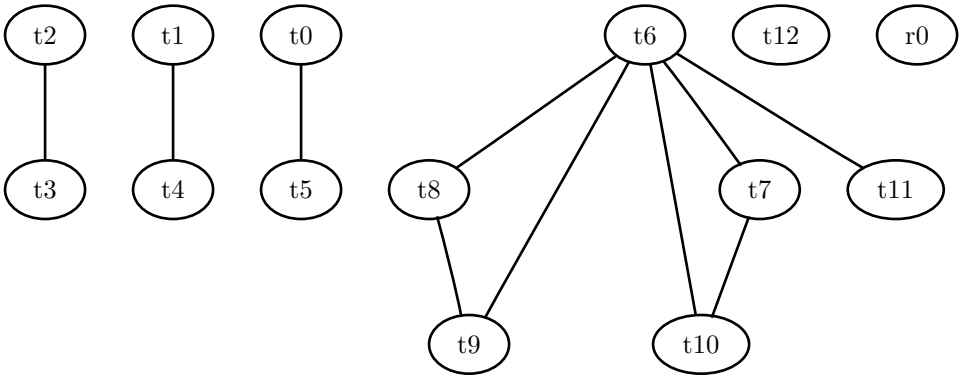
[2]Refer to the lecture notes for Lecture 3 at `http://www.cs.cmu.edu/~fp/courses/15411-f08/lectures/03-regalloc.pdf`.



(c) Reorder the statements so that the program gives the same result, but there is less interference (i.e., fewer registers are needed), or state that the instruction order is already optimal in this sense. If it is already optimal, you only need to show one answer each for parts (d) and (e).

The result in returned in `r0`.

Line	Live-in temps
`t2 <- 8;`	.
`t3 <- 2;`	`t2`
`t4 <- t2 - t3;`	`t2, t3`
`t1 <- 3;`	`t4`
`t5 <- t1 * t4;`	`t1, t4`
`t0 <- 7;`	`t5`
`t6 <- t0 + t5;`	`t0, t5`
`t8 <- 9;`	`t6`
`t9 <- 5;`	`t6, t8`
`t10 <- t8 + t9;`	`t6, t8, t9`
`t7 <- 4;`	`t6, t10`
`t11 <- t7 * t10;`	`t6, t7, t10`
`t12 <- t6 - t11;`	`t6, t11`
`return t12;`	`t12`
	`r0`



(d) Construct elimination orders using maximal cardinality search for each of the two interference graphs. If the algorithm has a choice, list the lowest-numbered temporary first.

Original

`t0, t1, t2, t3, t4, t5, t6, t7, t8, t9, t10, t11, t12`

Reordered

`t0, t5, t1, t4, t2, t3, t6, t7, t10, t8, t9, t11, t12`

(e) Use greedy coloring to allocate registers for both the original program and the reordered program. Label the registers `r0`, `r1`, ..., `r`n.

Original

Temp	Register
t0	r0
t1	r1
t2	r2
t3	r3
t4	r3
t5	r1
t6	r0
t7	r1
t8	r2
t9	r3
t10	r3
t11	r1
t12	r0

Reordered

Temp	Register
t0	r0
t5	r1
t1	r0
t4	r1
t2	r0
t3	r1
t6	r0
t7	r1
t10	r2
t8	r1
t9	r2
t11	r1
t12	r0

Problem 3 (20 points)

(a) Translate the program from the AST in problem 1(a) into an ABI-conformant[3] x86-64 assembly routine, selecting instructions and allocating the registers using an algorithm of your choosing. Describe briefly which algorithm you used. Use the general-purpose x86-64 registers[4] and beware the calling conventions. Assume that all arithmetic is done on signed 64-bit integers. It should be possible to assemble your functions using the GNU assembler. The function should be callable using the following C prototype:

[3]Refer to `http://www.x86-64.org/documentation/abi.pdf`.

[4]Refer to page 7 of `http://www.cs.cmu.edu/~fp/courses/15411-f08/misc/asm64-handout.pdf`.

```
extern long ass1a();
```

Instructions selection: Maximal munch similar to 1b, but using x86-64 instructions and loading constants into registers just before they are needed for an operation. Register allocation: Greedy coloring on simplicial elimination ordering as in 2e.

```
.globl ass1a
ass1a:
  movq $8, %rax
  movq $2, %rcx
  subq %rcx, %rax
  movq %rax, %rcx
  movq $3, %rax
  imulq %rax, %rcx
  movq $7, %rax
  addq %rcx, %rax
  movq $9, %rcx
  movq $5, %rdx
  addq %rcx, %rdx
  movq $4, %rcx
  imulq %rdx, %rcx
  subq %rcx, %rax
  ret
```

(b) Handwrite assembly code using only `mov`, `addq`, `imulq`, `subq`, and `ret` instructions which uses as few instructions as possible and carries out the required operations. You should not use any laws of modular arithmetic such as commutativity, associativity, or distributivity to optimize the computation, and you should not use constant folding, that is, all operations should be performed at runtime. It should be callable as

```
extern long ass1b();
```

```
.globl ass1b
ass1b:
  movq $5, %rdi
  addq $9, %rdi
  imulq $4, %rdi
  movq $8, %rax
  subq $2, %rax
  imulq $3, %rax
  addq $7, %rax
  subq %rdi, %rax
  ret
```

(c) How many instructions appear in the systematically constructed assembly code, and how many instructions does your hand-crafted code have? Explain why you believe your code to be optimal, under the given constraints. Do you believe your code also uses the minimal number of necessary registers? Explain why or why not.

The systematically generated code has 15 instructions. The hand-crafted code has 9 instructions. The hand-crafted code is optimal under the given constraints because at least one instruction is needed for each of the six arithmetic operations, one instruction is needed for the return, and one `mov` is needed to compute each of the two non-trivial terms (`3 * (8 - 2)` and `4 * (9 + 5)`).