Assignment 1
Instruction Selection and Register Allocation

15-411: Compiler Design
Miguel Silva (miguels@andrew)

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

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.

Problem 1 (20 points)

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

```
{ 1 - 2 + 3; }
```

would be

```
PLUS(MINUS(CONST(1),CONST(2)),CONST(3))
```

Here we use a term notation for trees where the node is the constructor and the subtrees are arguments. Use the names `PLUS`, `MINUS`, `TIMES`, and `DIV` for the constructors.

(i) `{ (1 + 2) *3; }`

(ii) `{ 7 - (1 - 2 *3) ); }`

(iii) `{ 7 + 10 + (1/2 *3); }`

(b) Translate the AST from problem 1(a)(c) into linear three-address form by applying maximal munch using the patterns in the table below. Subtrees should be translated left-to-right. Temporaries should be called `t0`, `t1`, ..., `t`n. A wildcard "_" in the pattern matches an arbitrary subtree.

Pattern	IR
`CONST(`c`)`	`t`i `<-` c
`DIV(_,CONST(`c`))`	`t`j `<-` `t`i `/ c`
`DIV(_,_)`	`t`j `<-` `t`i `/` `t`k
`PLUS(_,_)`	`t`i `<-` `t`j `+` `t`k
`MINUS(_,_)`	`t`i `<-` `t`j `-` `t`k
`TIMES(_,_)`	`t`i `<-` `t`j `*` `t`k

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

```
t0 <- 1
t1 <- 2
t2 <- t0 - t1
t3 <- 3
t4 <- t2 + t3
```

Problem 2 (20 points)

(a) Compute the live variables after each statement in the following programs (assuming some more complex instructions than before):

(i)
```
t1 <- 2
t2 <- 3
a <- t1 + t2
t3 <- a
t4 <- t2 - t1
t5 <- t4 + a + t3
```

(ii)
```
t1 <- 2
b <- 5
t2 <- 3
a <- 3 + b
a <- a + t2
t3 <- a
a <- 2 * a
b <- a * 3
t4 <- b + t1 + t3
```

(b) Construct the interference graph for both programs. Are they chordal? Justify your answers[1]

(c) Consider the following program

```
t1 <- 3
t2 <- 2
t3 <- 1
t4 <- t3 + 1
t5 <- t2 * 3
t6 <- t1 - 10
```

(i) Use the algorithm from Lecture 3 to allocate registers for the program above.

(ii) Does your allocation use the minimum number of registers for this interference graph?

(iii) Is it possible to reorder the statements so that the program gives the same result, but there is less interference (i.e., fewer registers are needed)? How many register do we need now?

[1] Refer to the lecture notes for Lecture 3 at `http://www.cs.cmu.edu/~fp/courses/15411-f09/lectures/03-regalloc.pdf`.

Problem 3 (20 points)

For this section, we are going to use the following assembly language:

r_i	register i, $0 \leq i < 3$
rr	register that holds a function's return value
ADD r_1 r_2 r_3	$r_3 \leftarrow r_1 + r_2$
SUB r_1 r_2 r_3	$r_3 \leftarrow r_1 - r_2$
MUL r_1 r_2 r_3	$r_3 \leftarrow r_1 * r_2$
DIV r_1 r_2 r_3	$r_3 \leftarrow r_1 / r_2$
MOVE r_1 r_2	$r_2 \leftarrow r_1$
LOADi c r_1	$r_1 \leftarrow c$, where c is a constant
Label:	identifies a program point
JUMP *label*	The execution of the program jumps to *label*
JUMPZERO r_i *label*	If r_0 is zero, the execution jumps to *label*. Otherwise, it resumes right after the JUMPZERO instruction

(a) Translate the following program into assembly, selecting instructions and allocating the registers.

```
a = 3;
b = 5;
return (a+b)*(a-b)*3;
```

(b) Using *Jump* and *JumpZero*, handwrite assembly code for `a =` e_1 `&&` e_2 and `a =` e_1 `||` e_2, where e_1 and e_2 are expressions. Assume that the variable `a` has been assigned register ra, that E_1 and E_2 stand for the assembly code corresponding to e_1 and e_2 respectively, and that both these expressions store their results in register re.

e_1 `&&` e_2	If e_1 is **FALSE**, then the result is **FALSE** (e_2 is not evaluated). Otherwise, the result is the result of e_2
e_1 `||` e_2	If e_1 is **TRUE**, then the result is the value of e_1 (e_2 is not evaluated). Otherwise, the result is the result of e_2
FALSE	The constant `0`
TRUE	Any non-zero value