CS 502 – Compiling and Programming Systems
Final and Qualifying Examination, 12/17/2009

Instructions: Read carefully through the whole exam first and plan your time. Note the relative weight of each question and part (as a percentage of the score for the whole exam). The total points is 100 (*ie*, your grade will be the percentage of your answers that are correct).

This exam is **open book, open notes**. You are free to refer to any book or other study materials you bring to the exam room.

You have **2 hours** to complete all five (5) questions. Write your answers on this paper (use both sides if necessary).

Name:

Student Number:

Signature

1. (LALR parsing; 10%) Consider the following augmented grammar:

$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & aBc \mid bCc \mid aCd \mid bBd \\ B & \rightarrow & e \\ C & \rightarrow & e \end{array}$$

   (a) (5%) Is this grammar LR(1)? Why or why not?
   Answer:

   Here are the LR(1) item sets:

$$\begin{array}{rlll} 0: & S' & \rightarrow \bullet S & \{\$\} \\ & S & \rightarrow \bullet aBc & \{\$\} \\ & S & \rightarrow \bullet bBd & \{\$\} \\ & S & \rightarrow \bullet aCd & \{\$\} \\ & S & \rightarrow \bullet bCc & \{\$\} \\ 1: & S & \rightarrow a \bullet Bc & \{\$\} \\ & S & \rightarrow a \bullet Cd & \{\$\} \\ & B & \rightarrow \bullet e & \{c\} \\ & C & \rightarrow \bullet e & \{d\} \\ 2: & S & \rightarrow b \bullet Bd & \{\$\} \\ & S & \rightarrow b \bullet Cc & \{\$\} \\ & B & \rightarrow \bullet e & \{d\} \\ & C & \rightarrow \bullet e & \{c\} \\ 3: & S & \rightarrow aB \bullet c & \{\$\} \\ 4: & S & \rightarrow aC \bullet d & \{\$\} \\ 5: & B & \rightarrow e \bullet & \{c\} \\ & C & \rightarrow e \bullet & \{d\} \\ 6: & S & \rightarrow bB \bullet d & \{\$\} \\ 7: & S & \rightarrow bC \bullet c & \{\$\} \\ 8: & B & \rightarrow e \bullet & \{d\} \\ & C & \rightarrow e \bullet & \{c\} \\ 9: & S & \rightarrow aBc \bullet & \{\$\} \\ 10: & S & \rightarrow aCd \bullet & \{\$\} \\ 11: & S & \rightarrow bBd \bullet & \{\$\} \\ 12: & S & \rightarrow bCc \bullet & \{\$\} \end{array}$$

   There are no conflicts so this grammar is LR(1).

   (b) (5%) Is this grammar LALR(1)? Why or why not?
   Answer:

   This grammar is not LALR(1). Consider merging states 5 and 8. This will introduce reduce-reduce conflicts.

2. (Semantics, code generation, run-time; 25%) Consider the Java **`synchronized`** statement:

> A **`synchronized`** statement acquires a mutual-exclusion lock on behalf of the executing thread, executes a block, then releases the lock. While the executing thread owns the lock, no other thread may acquire the lock.
>
> ```
> SynchronizedStatement:
>   synchronized (Expression) Block
> ```
>
> The type of *`Expression`* must be a reference type, or a compile-time error occurs.
>
> A **`synchronized`** statement is executed first by evaluating the *`Expression`*.
>
> If evaluation of the *`Expression`* completes abruptly for some reason, then the **`synchronized`** statement completes abruptly for the same reason.
>
> Otherwise, if the value of the *`Expression`* is **`null`**, a run-time error occurs.
>
> Otherwise, let the non-null value of the *`Expression`* be *v*. The executing thread acquires the lock associated with *v*. Then the *`Block`* is executed. If execution of the *`Block`* completes normally, then the lock is unlocked and the **`synchronized`** statement completes normally. If execution of the *`Block`* completes abruptly for any reason, then the lock is unlocked and the **`synchronized`** statement then completes abruptly for the same reason.
>
> Acquiring the lock associated with an object does not of itself prevent other threads from accessing fields of the object or invoking unsynchronized methods on theb object. Other threads can also use **`synchronized`** methods or the **`synchronized`** statement in a conventional manner to achieve mutual exclusion.
>
> The locks acquired by **`synchronized`** statements are the same as the locks that are acquired by **`synchronized`** methods. A single thread may hold a lock more than once.
>
> The example:
>
> ```
> class Test {
>   public static void main (String[] args) {
>     Test t = new Test();
>     synchronized (t) {
>       synchronized (t) {
>         System.out.println("made it!");
>       }
>     }
>   }
> ```
>
> prints `made it!`
>
> This example would deadlock if a single thread were not permitted to acquire a lock more than once.

Consider the steps needed to add the **synchronized** statement to MiniJava. Recall that statements causing abrupt completion include **break**, **continue**, and **return**. Also, assume that the MiniJava compiler and run-time library already supports the creation of threads, and that the run-time library contains support for a primitive `Lock` implementing the lock semantics described above (including that the same thread can reacquire the lock):

```
class Lock() {
  void acquire(); // Acquire this lock for the current thread.
  void release(); // Release this lock for the current thread.
}
```

(a) (5%) What changes are needed in the front-end (scanner/parser/type-checker)?
Answer:

New keyword **synchronized**.
Parsing and abstract syntax representing **synchronized** statements.
Type-checker must make sure *Expression* has reference type.

(b) (5%) What changes are needed in the run-time representation of arrays/objects?
Answer:

Every object needs potentially to associate a lock. We need to store sufficient information inside the object to obtain its lock. The simplest approach is to allocate a `Lock` instance for every allocated object, and store it in a `lock` field in the object header.

(c) (10%) Give a code template for translating **synchronized** blocks.
Answer:

```
  v = Expression;
  v.lock.acquire();
  Block;
  v.lock.release();
  goto exit;
breakLabel:
  v.lock.release();
  break;
continueLabel:
  v.lock.release();
  continue;
returnLabel:
  v.lock.release();
  return result;
exit:
```

Inside the *Block*, **break** translates to:

```
goto breakLabel;
```

continue generates:

```
goto continueLabel;
```

and **return** *[Expression]* generates:

```
[ result := Expression; ]
goto returnLabel;
```

for optional *Expression*.

(d) (5%) Assuming that most objects never need an associated lock (*ie*, they are never synchronized), what unnecessary overheads (in space and time) does adding support for **`synchronized`** statements impose on programs that never execute them? What about programs that run only one thread or never share references between threads?
Answer:

> We must associate a lock object with every object, even if that object is never synchronized. Single-threaded programs will pay the cost of lock acquire/release even though the locks are uncontended. Similarly for unshared references.

3. (Semantics, code generation, run-time; 10%) Consider an extension of Java that introduces the **`atomic`** statement. An **`atomic`** statement is similar to a **`synchronized`** statement:

   `atomic` *`Block`*

   except that it has no *`Expression`* with which to designate an associated lock.

   The **`atomic`** statements can be *simulated* by rewriting them at the source-code level as:

   `synchronized` `(Object.`**`class`**`)` *`Block`*

   That is, each **`atomic`** statement ensures mutual exclusion (*ie*, *serial* execution) with respect to *all* other **`atomic`** statements, *as if* they all used some unique global lock (in this case the unique lock associated with `Object.`**`class`**).

   Of course, implementing **`atomic`** statements in this way would impede thread concurrency. So long as an implementation preserves the *illusion* that **`atomic`** statements execute serially (one after another, and not overlapped) then that implementation is correct.

   The only way any pair of **`atomic`** statements executing concurrently (in different threads) can break the illusion of serial execution is if they *conflict*: the memory locations stored by one thread overlap with the memory locations loaded by the other thread. Architecture designers have proposed **`atomic`** hardware instructions that directly support this model:

   - `chkpt` *`fail_pc`*: begin logging memory loads/stores, on conflict discard memory stores and branch to *`fail_pc`*
   - `commit`: stop logging memory loads/stores

   For simplicity you may assume that *nested* pairs of `chkpt`/`commit` instructions are discarded by the hardware (*ie*, only top-level `chkpt`/`commit` pairs have any effect). Note that the hardware logs only memory and on conflict discards only memory stores.

   Describe how you would translate **`atomic`** statements. In particular, give a code template for translating **`atomic`** statemements.

   Answer:

   Translation is much the same as for **`synchronized`** statements, with `chkpt` /`commit` in place of `acquire`/`release`, except that we must save the live registers before the checkpoint, and restore them on failure.

```
  save registers
retry:
  chkpt fail
  Block
  commit
  goto exit
breakLabel:
  commit;
  break;
continueLabel:
  commit;
  continue;
returnLabel:
  commit;
  return result;
fail:
  restore registers
  goto retry;
exit:
```

4. (Liveness analysis and register allocation; 35%) Consider the following MiniJava program:

```
class Fact {
  static int f (int n) {
    if (n > 1) return Fact.f(n-1) * n;
    return 1;
  }
}
```

Assume that we are generating MIPS instructions, but for a machine with only two general-purpose registers:

- `$a0`: a caller-saved argument/result (live on exit) register
- `$s0`: a callee-save register

The MiniJava compiler generates the following MIPS instructions for the method `Fact.f`:

```
        move n $a0
L.5:
        bgt n 1 L.4
L.3:
        li t3 1
        move $a0 t3
        b L.0
L.4:
        subu t4 n 1
        move t1 t4
        move $a0 t1
        jal Fact.f
        move t2 $a0
        mul t5 t2 n
        move $a0 t5
        b L.0
L.6:
        b L.3
L.0:
```

(a) (Basic blocks, control flow graphs; 10%) Identify the *basic blocks* in this code, and draw its intraprocedural control flow graph (CFG) having nodes which are the *basic blocks* and edges representing control flow among them. For each *basic block*, summarize the temporaries/registers used by the block (before they are defined) and defined by the block. Annotate the edges of the CFG with the temporaries/registers live on that edge.

Answer:

```
0: def = n, use = $a0 ; goto 1 live = n
Fact.f:
        move n $a0
1: use = n ; goto 3 live = n, goto 2 live =
L.5:
        bgt n 1 L.4
2: def = (t3 $a0) ; goto 5 live = $a0
L.3:
        li t3 1
        move $a0 t3
        b L.0
3: def = t4 t1 $a0 t2 t5, use = n ; goto 5 live = $a0
L.4:
    subu t4 n 1
    move t1 t4
    move $a0 t1
    jal Fact.f
    move t2 $a0
    mul t5 t2 n
    move $a0 t5
    b L.0
4: goto 2 live = $a0
L.6:
    b L.3
5: use = $a0; live = $a0
L.0:
```

(b) (Interference graphs; 10%) Fill in the following adjacency table representing the interference graph for the method; an entry in the table should contain an $\times$ if the variable in the left column interferes with the corresponding variable/register in the top row. Since machine registers are pre-colored, we choose to omit adjacency information for them. Naturally, you must still record if a non-precolored node interferes with a pre-colored node; the columns for pre-colored nodes are there for that purpose.

Also, record the *unconstrained* move-related nodes in the table by placing an $\circ$ in any empty entry where the variable in the left column is the source or target of any move involving the variable/register in the top row. **Remember that nodes that are move-related should not interfere if their live ranges overlap only starting at the move**.

	`$s0`	`$a0`	`t1`	`t2`	`t3`	`t4`	`t5`	`n`
`t1`								
`t2`								
`t3`								
`t4`								
`t5`								
`n`								

Answer:

	`$s0`	`$a0`	`t1`	`t2`	`t3`	`t4`	`t5`	`n`
`t1`		$\circ$				$\circ$		$\times$
`t2`		$\circ$						$\times$
`t3`		$\circ$						
`t4`			$\circ$					$\times$
`t5`		$\circ$						
`n`		$\times$	$\times$	$\times$		$\times$		

(c) (Register allocation; 10%) Show the steps of a coalescing graph-coloring register allocator as it assigns registers to the variables in the method. Use the *George criterion* for coalescing nodes: node *a* can be coalesced with node *b* only if all significant-degree (*ie*, degree >= K) neighbors of a already interfere with b. Show the final method, noting any redundant moves.

Answer:

i. No non-move-related low-degree node

ii. Coalesce `t5` with `$a0`

	`$s0`	`$a0`	`t1`	`t2`	`t3`	`t4`	`n`
`t1`		◦				◦	×
`t2`		◦					×
`t3`		◦					
`t4`			◦				×
`n`		×	×	×		×	

iii. Coalesce `t4` with `t1`

	`$s0`	`$a0`	`t1`	`t2`	`t3`	`n`
`t1`		◦				×
`t2`		◦				×
`t3`		◦				
`n`		×	×	×		

iv. Coalesce `t3` with `$a0`

	`$s0`	`$a0`	`t1`	`t2`	`n`
`t1`		◦			×
`t2`		◦			×
`n`		×	×	×	

v. Coalesce `t2` with `$a0`

	`$s0`	`$a0`	`t1`	`n`
`t1`		◦		×
`n`		×	×	

vi. Coalesce `t1` with `$a0`

	`$s0`	`$a0`	`n`
`n`		×	

vii. Simplify `n`

viii. Select: `n`=`$s0`, `t1`/`t2`/`t3`/`t4`/`t5`=`$a0`

ix. Result:

```
      move $s0 $a0
L.5:
      bgt $s0 1 L.4
L.3:
      li $a0 1
# move $a0 $a0
      b L.0
L.4:
      subu $a0 $s0 1
# move $a0 $a0
# move $a0 $a0
      jal Fact.f
# move $a0 $a0
      mul $a0 $a0 $s0
# move $a0 $a0
      b L.0
L.6:
      b L.3
L.0:
```

(d) (Procedure prologue/epilogue; 5%) The code so far does not include a prologue or epilogue. Write the prologue and epilogue for this method. Remember to save/restore any callee-save registers it defines.
Answer:

Prologue:

```
sw $ra -8($sp)
sw $s0 -4($sp)
subu $sp Fact.f.framesize
```

Epilogue:

```
addu $sp Fact.f.framesize
lw $s0 -4($sp)
lw $ra -8($sp)
j $ra
```

5. (A challenge!; 20%) The Java Language Specification requires that every program pass a definite assignment analysis to make sure that every local variable has been *definitely assigned* before it is accessed; otherwise a compile-time error must occur.

The idea behind *definite assignment* is that an assignment to the local variable must occur on every execution path to the access. The analysis takes into account the structure of statements and expressions; it also provides a special treatment of the expression operators `!`, `&&`, , and `? :`, and of **`boolean`**-valued constant expressions.

For example, a Java compiler recognizes that `k` is definitely assigned before its access (as an argument of a method invocation) in the code:

```
{
  int k;
  if (v > 0 && (k = System.in.read()) >= 0)
    System.out.println(k);
}
```

because the access occurs only if the value of the expression:

```
v > 0 && (k = System.in.read()) >= 0
```

is **`true`**, and the value can be **`true`** only if the assignment to `k` is executed (more properly, evaluated).

Similarly, a Java compiler will recognize that in the code:

```
{
  int k;
  while (true) {
    k = n;
    if (k >= 5) break;
    n = 6;
  }
  System.out.println(k);
}
```

the variable `k` is definitely assigned by the **`while`** statement because the condition expression **`true`** never has the value **`false`**, so only the **`break`** statement can cause the **`while`** statement to complete normally, and `k` is definitely assigned before the **`break`** statement. On the other hand, the code:

```
{
  int k;
  while (n < 4) {
    k = n;
    if (k >= 5) break;
    n = 6;
  }
  System.out.println(k);  // k is not definitely assigned before this
}
```

must be rejected by a Java compiler, because in this case the **`while`** statement is not guaranteed to execute its body as far as the rules of definite assignment are concerned.

Except for the special treatment of the conditional boolean operators `&&`, , and `? :` and of **`boolean`**-valued constant expressions, the values of expressions are not taken into account in the flow analysis.

For example, a Java compiler must produce a compile-time error for the code:

```
{
  int k;
  int n = 5;
  if (n > 2)
    k = 3;
  System.out.println(k);  // k is not definitely assigned before this
}
```

even though the value of `n` is known at compile time, and in principle it can be known at compile time that the assignment to `k` will always be executed (more properly, evaluated). A Java compiler must operate according to rules that recognizes only constant expressions; in this example, the expression `n>2` is not a constant expression.

As another example, a Java compiler will accept the code:

```
void flow(boolean flag) {
  int k;
  if (flag)
    k = 3;
  else
    k = 4;
  System.out.println(k);
}
```

as far as definite assignment of `k` is concerned, because the rules allow it to tell that `k` is assigned no matter whether the `flag` is **`true`** or **`false`**. But the rules do not accept the variation:

```
void flow(boolean flag) {
  int k;
  if (flag)
  k = 3;
  if (!flag)
    k = 4;
  System.out.println(k); // k is not definitely assigned before here
```

and so compiling this program must cause a compile-time error to occur.

You may find the following definitions useful:

before — variables definitely assigned before evaluation of a given statement or expression

after — variables definitely assigned after evaluation of a given statement or expression, assuming it completes normally (*ie*, not abruptly)

vars — all variables available in the scope of a given statement or expression

true — variables definitely assigned after evaluation of a given expression, assuming the expression evaluates to **`true`**

false — variables definitely assigned after evaluation of a given expression, assuming the expression evaluates to **`false`**

as you consider the data flow equations that arise for each syntactic statement or expression in Java.

Sketch how to perform definite assignment analysis for MiniJava (*ie*, ignore constructs for exceptions: **`try`**, **`throw`**, **`catch`**). Initially, ignore the difficulties posed by statements that cause *abrupt completion* of their enclosing statement: **`break`**, **`continue`**, and **`return`**. Then, describe how these can be incorporated into your analysis by considering what implications they have for the flow nodes that they target.

Answer:

See `http://en.wikipedia.org/wiki/Definite_assignment_analysis`.