

CS 502 – Compiling and Programming Systems

Mid-term Examination, 10/30/07

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

Name:

Student Number:

Signature

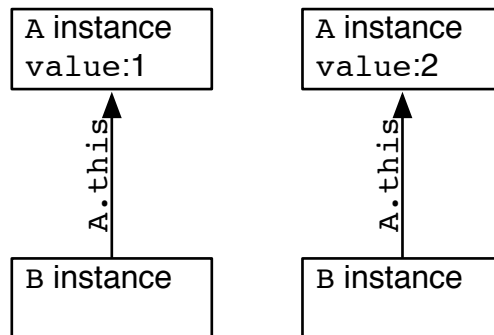
1. (MiniJava front-end; 20%) Consider the changes you would need to make to each phase of the MiniJava compiler front-end in order to add support for Java *inner classes*, which are introduced to the MiniJava grammar as follows:

```
<Class> ::= class <Id> [ extends <Id> ] { <Member>* }
<Member> ::= <Field> | <Method> | <Class>
```

where the only change is to permit <Class> to appear as a member. The rules for <Class>, <Field>, and <Method> are as before. Every instance of an inner class is scoped inside the instance of the outer class in which it is instantiated. Consider the following Java program:

```
class A {
    int value;
    class B {
        B() { System.out.println(A.this.value); }
    }
    A(int i) {
        this.value = i;
        new B();
    }
    public static void main (String[] args) {
        new A(1);
        new A(2);
    }
}
```

Here, two instances of A are constructed, each of which has a single inner B instance. Picture this as follows:



The syntax `A.this` allows each B instance to refer to its enclosing A instance. Thus, running this Java program results in the output:

```
1
2
```

Describe, specifically, what you must change in each of the following phases of the MiniJava compiler to support inner classes:

(a) (Scanning; 5%)

Answer:

No changes.

(b) (Parsing; 5%)

Answer:

Change syntax for `<Var>` to permit `.this` to be applied to a name like any other field access. Also, change syntax to permit a `<ClassDec>` as a `<MemberDec>`.

(c) (Semantic processing; 5%)

Answer:

Resolve `.this` when qualifying a class name to refer to the instance type of the class. First, check to make sure that the named class is an enclosing class for the occurrence of `.this`.

(d) (Translation to IR trees; 5%)

Answer:

When allocating an instance of inner class, capture the enclosing instance as a hidden `.this` field. When using `.this`, follow as many of these hidden links as represented by the depth of nesting from the enclosing class.

2. (Run-time representations; 10%) Bonus for those awake in class! Java's inner classes are an example of *lexical* scoping. In a functional programming language, one might write the following analogous program:

```
function A(int value) {  
    function B() { println(value); }  
    B();  
}
```

A(1);
A(2);

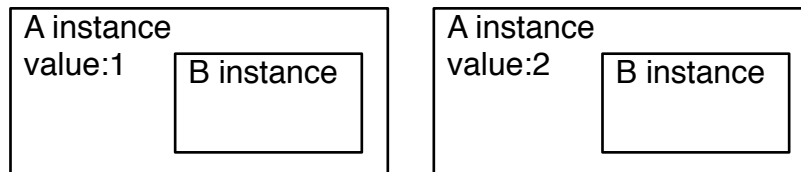
which produces the same output as the Java program, except that in this program the call frames for each activation of functions A (which includes storage for `value`) and B can be allocated on a stack, whereas in the Java program the instances of classes A and B are usually allocated in the (garbage collected) dynamic allocation heap.

- (a) (5%) Describe a mechanism by which function B can refer to the appropriate value of function A.

Answer:

Static chains: Calls to nested functions must pass their outer scope's static link (FP to most recent activation of outer-scope function) as one of the arguments to the call.

- (b) (5%) In the Java program above, it is possible to avoid allocating each B instance separately in the heap (so as to reduce work for the garbage collector), instead physically *inlining* each B instance inside its containing A instance, like this:



Why can Java *not* generally allocate storage for inner instances inside their containing instance?

Answer:

Inner objects can escape – they can be returned to dynamic scopes outside the outer object. This allows them to outlive the outer object.

3. (Context free grammars, LL parsing; 20%) Consider the following simple context free grammar:

$$\begin{aligned}
 S &\rightarrow U;W \\
 U &\rightarrow Ua \\
 U &\rightarrow \epsilon \\
 W &\rightarrow bW \\
 W &\rightarrow b
 \end{aligned}$$

(a) (10%) Modify this grammar so that it can be parsed by an LL(1) top-down parser.

Answer:

$$\begin{aligned}
 S &\rightarrow U;W \\
 U &\rightarrow U' \\
 U' &\rightarrow aU' \\
 U' &\rightarrow \epsilon \\
 W &\rightarrow bW' \\
 W' &\rightarrow W \\
 W' &\rightarrow \epsilon
 \end{aligned}$$

(b) (10%) Give the LL(1) parse table for the modified grammar.

Answer:

	;	a	b	\$
<i>S</i>	$S \rightarrow U;W$	$S \rightarrow U;W$		
<i>U</i>	$U \rightarrow U'$	$U \rightarrow U'$		
<i>U'</i>	$U' \rightarrow \epsilon$	$U' \rightarrow aU'$		
<i>W</i>			$W \rightarrow bW'$	
<i>W'</i>			$W' \rightarrow W$	$W' \rightarrow \epsilon$

4. (Context free grammars, LR parsing; 20%) Consider the following simple context free grammar:

$$\begin{aligned} S &\rightarrow A\$ \\ A &\rightarrow \epsilon \\ A &\rightarrow yA \\ A &\rightarrow yAz \end{aligned}$$

- (a) (5%) Is this grammar LR(0)? Why or why not?

Answer:

No: having parsed tokens reducing to A, there will be a shift-reduce conflict between productions $A \rightarrow yA$ and $A \rightarrow yAz$.

- (b) (10%) *Exhibit* the LR(1) state machine for this grammar. *Indicate* any and all conflicts. Is the grammar LR(1)?

Answer:

$$\begin{aligned} I_0 : & & S &\rightarrow \bullet A, & \$ \\ & & A &\rightarrow \bullet, & \$ \\ & & A &\rightarrow \bullet yA, & \$ \\ & & A &\rightarrow \bullet yAz, & \$ \\ I_1 = \text{goto}(I_0, A) : & & S &\rightarrow A \bullet, & \$ \\ I_2 = \text{goto}(I_0, y) : & & A &\rightarrow y \bullet A, & \$ \\ & & A &\rightarrow y \bullet Az, & \$ \\ & & A &\rightarrow \bullet, & \$z \\ & & A &\rightarrow \bullet yA, & \$z \\ & & A &\rightarrow \bullet yAz, & \$z \\ I_3 = \text{goto}(I_2, A) : & & A &\rightarrow yA \bullet, & \$ \\ & & A &\rightarrow yA \bullet z, & \$ \\ I_4 = \text{goto}(I_2, y) = \text{goto}(I_4, y) : & & A &\rightarrow y \bullet A, & \$z \\ & & A &\rightarrow y \bullet Az, & \$z \\ & & A &\rightarrow \bullet, & \$z \\ & & A &\rightarrow \bullet yA, & \$z \\ & & A &\rightarrow \bullet yAz, & \$z \\ I_5 = \text{goto}(I_3, z) : & & A &\rightarrow yAz \bullet, & \$ \\ I_6 = \text{goto}(I_4, A) : & & A &\rightarrow yA \bullet, & \$z \\ & & A &\rightarrow yA \bullet z, & \$z \\ I_7 = \text{goto}(I_6, z) : & & A &\rightarrow yAz \bullet, & \$z \end{aligned}$$

There is a shift-reduce conflict on lookahead z in I_6 between $A \rightarrow yA \bullet$ and $A \rightarrow yA \bullet z$, so the grammar is not LR(1).

- (c) (5%) For what (if any) k is this grammar LR(k)? Why? (Hint: consider the possible input string $y y z$ and its possible parses.)

Answer:

None, since the grammar is ambiguous.

5. (Control flow graphs, liveness analysis, register allocation; 30%) Consider the following MiniJava program to compute factorial:

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

This program has been compiled for a machine with 2 general-purpose registers:

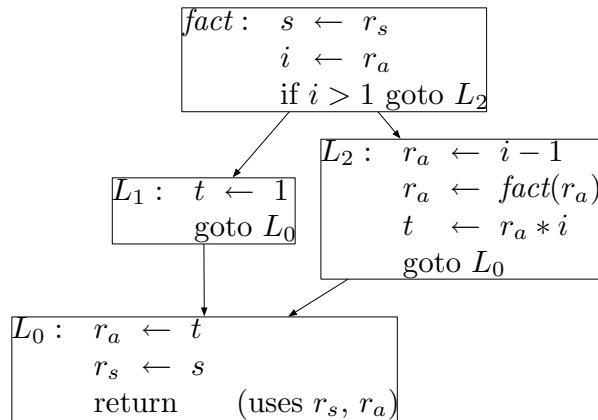
- r_s : a callee-save register
- r_a : a caller-save argument/result register

as follows:

```
fact:  $s \leftarrow r_s$ 
         $i \leftarrow r_a$ 
        if  $i > 1$  goto  $L_2$ 
 $L_1$ :  $t \leftarrow 1$ 
        goto  $L_0$ 
 $L_2$ :  $r_a \leftarrow i - 1$ 
         $r_a \leftarrow \text{fact}(r_a)$ 
         $t \leftarrow r_a * i$ 
        goto  $L_0$ 
 $L_0$ :  $r_a \leftarrow t$ 
         $r_s \leftarrow s$ 
        return          (uses  $r_s, r_a$ )
```

- (a) (5%) Draw the control flow graph for this program, with nodes that are the program's *basic blocks* (ie, not individual instructions) and with edges representing the flow of control between the basic blocks.

Answer:



- (b) (10%) Annotate each *instruction* with the variables/registers live-out at that instruction.

	Def	Use	LiveOut
<i>fact</i> : $s \leftarrow r_s$			
$i \leftarrow r_a$			
if $i > 1$ goto L_2			
L_1 : $t \leftarrow 1$			
goto L_0			
L_2 : $r_a \leftarrow i - 1$			
$r_a \leftarrow fact(r_a)$			
$t \leftarrow r_a * i$			
goto L_0			
L_0 : $r_a \leftarrow t$			
$r_s \leftarrow s$			
return		r_s, r_a	

Answer:

	Def	Use	LiveOut
<i>fact</i> : $s \leftarrow r_s$	s	r_s	s, r_a
$i \leftarrow r_a$	i	r_a	s, i
if $i > 1$ goto L_2		i	s, i
L_1 : $t \leftarrow 1$	t		s, t
goto L_0			s, t
L_2 : $r_a \leftarrow i - 1$	r_a	i	s, i, r_a
$r_a \leftarrow fact(r_a)$	r_a	r_a	s, i, r_a
$t \leftarrow r_a * i$	t	r_a, i	s, t
goto L_0			s, t
L_0 : $r_a \leftarrow t$	r_a	t	s, r_a
$r_s \leftarrow s$	r_s	s	r_s, r_a
return		r_s, r_a	

- (c) (5%) Fill in the following adjacency table representing the interference graph for the program; 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.**

	r_s	r_a	s	t	i
s					
t					
i					

Answer:

	r_s	r_a	s	t	i
s	\circ	\times		\times	\times
t		\circ	\times		
i		\times	\times		

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

Answer:

- i. Coalesce t with r_a :

	r_s	r_a	s	i
s	\circ	\times		\times
i		\times	\times	

- ii. Possible spill s :

	r_s	r_a	i
i		\times	

- iii. Simplify i :
 iv. Color $i \equiv r_s$
 v. Actual spill s

vi. Retain coalesces from before first spill, rewrite program, recompute liveness:

	Def	Use	LiveOut
<i>fact</i> : $\mathcal{M}[s_{loc}] \leftarrow r_s$		r_s	r_a
$i \leftarrow r_a$	i	r_a	i
if $i > 1$ goto L_2		i	i
L_1 : $r_a \leftarrow 1$	r_a		r_a
goto L_0			r_a
L_2 : $r_a \leftarrow i - 1$	r_a	i	i, r_a
$r_a \leftarrow fact(r_a)$	r_a	r_a	i, r_a
$r_a \leftarrow r_a * i$	r_a	r_a, i	r_a
goto L_0			r_a
L_0 : $r_s \leftarrow \mathcal{M}[s_{loc}]$	r_s		r_s, r_a
return		r_s, r_a	

	r_s	r_a	i
i		×	

- vii. Simplify i
- viii. Color $i \equiv r_s$
- ix. Final program:

```

fact:  $\mathcal{M}[s_{loc}] \leftarrow r_s$ 
         $r_s \leftarrow r_a$ 
        if  $r_s > 1$  goto  $L_2$ 
     $L_1$ :  $r_a \leftarrow 1$ 
        goto  $L_0$ 
     $L_2$ :  $r_a \leftarrow r_s - 1$ 
         $r_a \leftarrow fact(r_a)$ 
         $r_a \leftarrow r_a * r_s$ 
        goto  $L_0$ 
     $L_0$ :  $r_s \leftarrow \mathcal{M}[s_{loc}]$ 
        return

```