Compilers Comprehensive Exam
Fall 2003

This is a 30 minute, closed book exam. Please mark your answers in the blue book.

1. (5 points) Assume we have a statically typed language with polymorphic types and type inference (in the style of ML or Haskell). In one or two sentences explain how a self-application, such as

   $$f(f)$$

   is typed. Depending on what assumptions you make, you can answer this question either so that type inference succeeds or that it fails, but in either case you should pinpoint why it succeeds or fails.

```
Since f is a function, it has type a -> b for some unknowns a and b.
Because f is also the argument, f must have type a.  The type equation
   a -> b = a
has no finite solution, but does have infinite solutions, which correspond
to unification with the occurs check (for finite solutions) and without
(for infinite solutions).
```

2. (5 points) Consider the following nested loops (written in C). In one sentence give one reason an optimizer might choose to transform the first loop nest into the second. In one sentence give one reason an optimizer might choose to transform the second loop nest into the first.

```
for(i = 0; i < a, i++)
    for(j = 0; j < b; j++)
        A[i][j] = A[i][j+1] * 2;
```

```
for(j = 0; j < b; j++)
    for(i = 0; i < a, i++)
        A[i][j] = A[i][j+1] * 2;
```

```
The first nested loop has the better cache performance, as memory
elements are accessed in order.  The second nested loop has no
dependencies between iterations of the inner loop, and so is good for
any number of other optimizations (e.g., instruction scheduling).
```

3. (6 points) Consider the following flex-like specification. Parentheses are used to show the association of operations and are not part of the input alphabet.

aa^*	`{ return Token1; }`
$c(a\|b)^*$	`{ return Token2; }`
ab^*c	`{ return Token3; }`
caa^*	`{ return Token4; }`
$b^*aa^*(c\|\epsilon)$	`{ return Token5; }`

Show how the following string is partitioned into tokens. Label each lexeme with the integer of the correct token class.

abcabcaabbaacccabaccbb

```
abc abc aa bbaac c caba c cbb
 3   3   1   5   2  2   2  2
```

4. (4 points) In one or two sentences explain why an LR(1) parser can handle the following grammar while a LL(1) parser cannot:

$$\begin{aligned} \text{Loop} \rightarrow\ & \text{do stmt while expr} \\ |\ & \text{do stmt until expr} \\ |\ & \text{do stmt forever} \end{aligned}$$

```
A top-down parser must decide which production to use when it sees the
terminal 'do', whereas a bottom-up parser makes this decision only
after the entire right-hand side of the production is on the stack.
```

5. (10 points) Below are the "action" and "goto" tables for an LR parser. The "goto" table includes only moves of the parsing automaton on non-terminals; the moves on terminals are encoded in the shift moves of the "action" table. The actions should be interpreted as follows:

- $s(n)$ shifts the input and goes to state n.
- $r(n, T)$ pops n elements off of the stack and pushes the non-terminal T onto the stack. An r-action is a reduce move, given in a non-standard way.
- *acc* means accept.
- A blank is an error entry.

The non-terminals of the grammar from which these tables were generated are A, B, and C. No two productions for A have the same number of symbols on the right-hand side; similarly, all productions for B and C have different lengths.

What is the grammar from which these tables were produced?

State	*action*						*goto*		
	a	b	c	d	e	$	A	B	C
0	s(5)			s(4)			1	2	3
1		s(6)				acc			
2		r(1,A)	s(7)		r(1,A)	r(1,A)			
3		r(1,B)	r(1,B)		r(1,B)	r(1,B)			
4	s(5)			s(4)			8	2	3
5		r(1,C)	r(1,C)		r(1,C)	r(1,C)			
6	s(5)			s(4)				9	3
7	s(5)			s(4)					10
8		s(6)			s(11)				
9		r(3,A)	s(7)		r(3,A)	r(3,A)			
10		r(3,B)	r(3,B)		r(3,B)	r(3,B)			
11		r(3,C)	r(3,C)		r(3,C)	r(3,C)			

```
A -> A b B
A -> B
B -> B c C
B -> C
C -> d A e
C -> a
```

The essence of the problem is to discover what can be on the stack when a reduction is about to happen. One way to solve the problem is to reconstruct the parsing DFA from the table and read the moves. A simpler way is to reason as follows. In state 9 there is a reduce move r(3,A), so we know there is a production A -> XYZ for some X, Y, and Z. How could the DFA get into state 9? It could get there from a 'goto B' out of state 6. Therefore, Z = B. One way to get to state 6 is via a 'shift b' action from state 8, so Y = b. State 8 is reached from a goto on a reduce to A, so X = A and the production is A -> AbB.

The reasoning for the other productions is similar.