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.

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;
```

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.

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

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

`abcabcaabbaacccabaccbb`

4. (4 points) In one or two sentences explain why a bottom-up parser can handle the following grammar while a top-down parser cannot:

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

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)			