CS 453 Introduction to Compilers	**Midterm Examination**
Spring 2009	**March 12, 2009**

75 minutes (maximum)
Closed Book

- You may use one side of one sheet (8.5x11) of paper with any notes you like.
- This exam has 9 pages, including this cover page and a blank page at the end. Do all your work on these exam sheets, use the backs of the pages if needed.
- Be specific and clear in your answers. If there is any question about what is being asked, then indicate the assumptions you need to make to answer the question.
- Show all your work if you wish to be considered for partial credit.

Question	Points	Score
1	20	
2	20	
3	20	
4	20	
5	10	
6	10	

Name: _____

Email: _____

DO NOT TURN TO NEXT PAGE TILL YOU GET PERMISSION

1. [20 points] Expressions, Assignments, and Printlns in the MiniJava Compiler.
   Given the following set of statements,

```
{
  a = 2;
  System.out.println( 7 + 1 * a );
}
```

   (a) draw the implicit parse tree,

   (b) draw the AST,

   (c) show the output based on an interpretation of the AST, and

   (d) write a MIPS main routine that performs the expressed computation assuming that the function printint is available. You can also write a comment to indicate what is being done once you have shown the MIPS code for a similar set of operations. For example, once you have pushed a constant value onto the stack, later instances of that operation can show up as the comment "# push 5 onto stack".

problem 1 cont ...

2. [20 points] Predictive Parse Table.
   Assuming that NUM, EOF, and REAL are all tokens, show the nullable property and the FIRST and FOLLOW sets for all of the nonterminals in the following grammar, which is a common input format for sparse matrices:

```
(1) matrix -> NUM entry_list EOF
(2) entry_list -> entry entry_list
(3) entry_list -> epsilon
(4) entry -> col row val
(5) col -> NUM
(6) row -> NUM
(7) val -> REAL
```

   Using the FIRST and FOLLOW sets, construct the predictive parse table.

3. [20 points] Top-Down Parsing. For the following grammar, construct the recursive-descent parser functions for the nonterminals matrix and entry_list. Use panic mode error handling.

```
(1) matrix -> NUM entry_list EOF
(2) entry_list -> entry entry_list
(3) entry_list -> epsilon
(4) entry -> NUM NUM REAL
```

The grammar is a simplified version of the grammar in question 2.

You can assume the functions match() and panic() are available.

```
match(tok) { if(tok=lookahead) lookahead = scan();
             else throw new SyntaxException(message); }
panic( nonterminal ) {
    print error;
    while ( scan() not in (FOLLOW(nonterminal)) ) {
    }
}
```

4. [20 points] Using JavaCUP.
   Assume a small scripting language that manipulates integers. The scripting language includes a made up operator $\oplus$ that performs kiddition. When performing kiddition on two integers, the result is the integer value for the string resulting from concatenating the string representations for the two integers. For example, $42 \oplus 987$ evaluates to the integer 42987.

   The scripting language also includes the made up operator $\odot$, which given any two integer operands always results in the value 1. For example, $42 \odot 987$ evaluates to the integer 1.

   Assume that $\oplus$ is right associative, and $\odot$ is left associative. Also assume that $\odot$ has higher precedence than $\oplus$.

   (a) What does the expression $5 \oplus 67 \oplus 6 \odot 789 \odot 1234$ evaluate to?

   (b) Write a grammar for this language and use the JavaCUP precedence keyword to resolve ambiguity. You do not have to create a text version of the operators. Just use the symbols $\oplus$ and $\odot$ as terminals.

   (c) Associate actions with the grammar rules that evaluate the expression being parsed. Assume that the type for the integer literal token NUM is a Java Integer. Recall that the Integer type has a toString() method and that there is an Integer.parseInt() method.

5. [10 points] Syntax-Directed Translation.
   Write a syntax directed definition that creates a list where the identifiers are listed in reverse order of the list of identifiers being parsed. There is more than one way to do this. Assume that the ID token is of type String and use the following grammar:

```
(1) id_list -> id_list      ID
```

```
(2) id_list -> /* epsilon */
```

6. [10 points] lvalues and rvalues.
   Categorize all of the subexpressions in the following C program statement as lvalues being used as rvalues, lvalues being used as lvalues, or strictly rvalues.

```
*(p+1) = 45 * (*q) + t->f
```

rvalue	lvalue used as rvalue	lvalue

(empty page)