CSE P 501 Exam 11/17/05 Sample Solution

1. (8 points) Write a regular expression or set of regular expressions that generate the following sets of strings. You can use abbreviations (i.e., name = regular expression) if it helps to make your answer clearer.

(a) (4 points) All strings of a's, b's, and c's that contain an even number of a's (if they contain any a's at all).

(b|c)* | ((b|c)* a (b|c)* a (b|c)*)*

(b) (4 points) All strings of a's, b's, c's, A's, B's, and C's that follow the Java capitalization convention for variable identifiers. This convention is that an identifier must begin with a lower-case letter (abc), followed by 0 or more other letters, with the restriction that no two capital letters (ABC) may be next to each other. So a, aA, aAaB, aaCabcBa, and abc are all examples of properly capitalized identifiers, while A, Aa, aBC, and aBBa are examples of identifiers that are not properly capitalized.

Here's one solution:

(a|b|c)+ ((A|B|C) (a|b|c)+)* (A|B|C)?

Note that there needs to be at least one leading lower case letter according to the definition, so expressions that can derive an empty string did not receive full credit. Also, it is possible for an identifier to end with a capital letter.

2. (8 points) For each of the following regular expressions, give a brief description in English of the set of strings generated by the expression. (Remember that * has higher precedence – binds tighter – than concatenation.)

(Note: The question asked for a description of the sets of strings generated, not just a paraphrase or transliteration of the equations into English. Answers that did the later lost a point.)

(a) (4 points) a*ba*ba*ba*

All strings of a's and b's containing exactly 3 b's.

(b) (4 points) (x*y*)* xx (x | y)*

All strings of x's and y's with at least one pair of adjacent x's.

3. (5 points) The C family of languages includes prefix and binary + and – operators, as well as the increment and decrement operators ++ and --, which can appear either before or after an expression. Recalling that a scanner uses a principle of longest match when it's reading the input and parsing it into tokens, what are the tokens in the following statement?

```
a++=++thing+++other--;
```

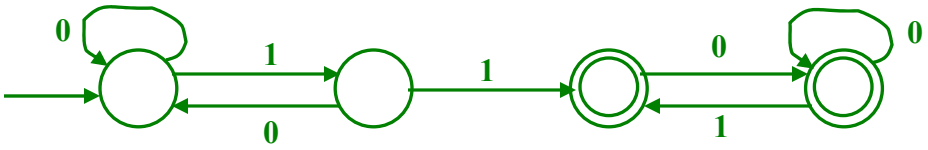
(Notes: Don't worry if this statement is not legal C – after all, the scanner doesn't care, does it? Be sure that your tokens are easy to understand: PLUS, PLUSPLUS and ID(xyzzy) are easy to understand; FOO, CAT, and THING are not.)

ID(a) PLUSPLUS EQUALS PLUSPLUS ID(thing) PLUSPLUS PLUS ID(other) MINUSMINUS SCOLN

4. (8 points) The follow regular expression is alleged to generate all strings of 0's and 1's that contain exactly one pair of adjacent 1's:

(0 | 10)* 11 (0 | 01)*

Draw a DFA that accepts strings generated by this regular expression. Be sure that your diagram distinguishes between final and non-final states (use a double circle for final states and a single circle for non-final states). You don't need to use a formal NFA to DFA construction, just draw a DFA that accepts this set of strings.



5. (3 points) The suggested strategy for checking types and other static semantics in MiniJava is to make at least two passes over the AST – the first to collect global information about the types (classes) defined in the program, and later passes to process the information (including methods) in each class.

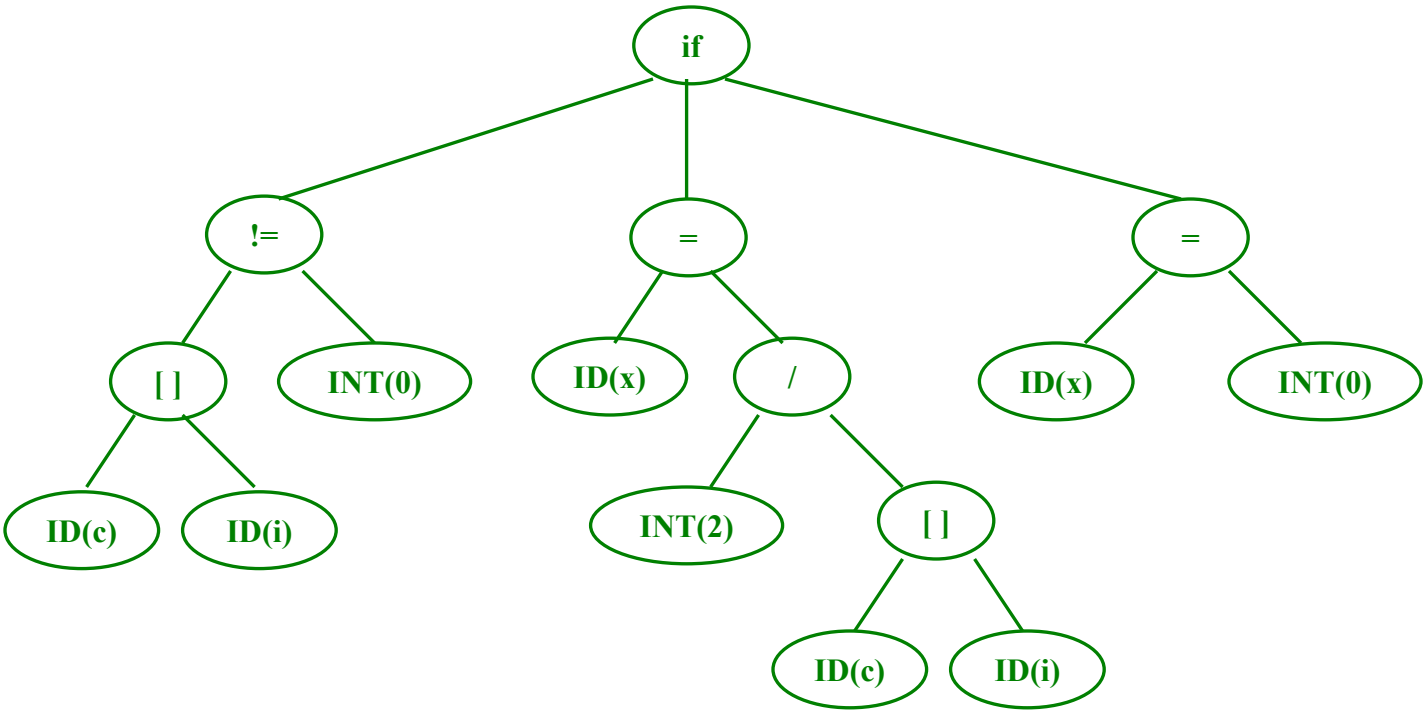
Give one technical reason why we need multiple passes over the AST. Why can't we do all of the type checking in a single traversal of the AST?

Class and method names can be used before they appear in the source program, so we either need two passes, or we need some way of backing up to finish processing earlier uses of items declared later in the program, which amounts to the same thing.

6. (8 points) Consider the following C (or Java, C++, etc.) program fragment.

```
if (c[i] != 0)
   x = 2 / c[i];
else
   x = 0;
```

Draw an abstract syntax tree (AST) for this program fragment. The AST should reflect the semantics of the source program, for example, the abstract syntax for `c[i]` should be shown as an array subscripting operation, rather than breaking it down into a sequence of address arithmetic expressions, or, at the other extreme, just leaving it as a single node containing the label `c[i]`.



7. (8 points) Consider the following grammar:

exprs ::= *exprs* + *expr* | *exprs* * *expr* | *expr*
expr ::= x

(a) (4 points) Is this grammar ambiguous or unambiguous? If it's ambiguous, show that by giving two different parse trees for some string generated by the grammar. If it's not ambiguous, give an informal argument why it is not.

This is not ambiguous. We could prove this formally, but informally, the only way to derive a sequence of operations is to expand the leftmost nonterminal *exprs* repeatedly and, since this can only derive a single *expr* or another *exprs* that is also left-recursive, we have no choice about which alternative for *exprs* to expand at any point.

(b) (4 points) Does this grammar properly capture the normal precedence of arithmetic operators * and +? (i.e., * should have higher precedence than +) If so, give a brief argument as to why it does, if not, give an example that shows where it fails to capture the correct precedence relationship.

No. Consider the string x + x * x. The only possible derivation for this is

***exprs* -> exprs * expr -> (exprs + expr) * expr -> … -> (x + x) * x**

Since all derivations in this grammar are left associative, with no distinction between + and *, the operator + binds more tightly than * in the expression x+x*x.

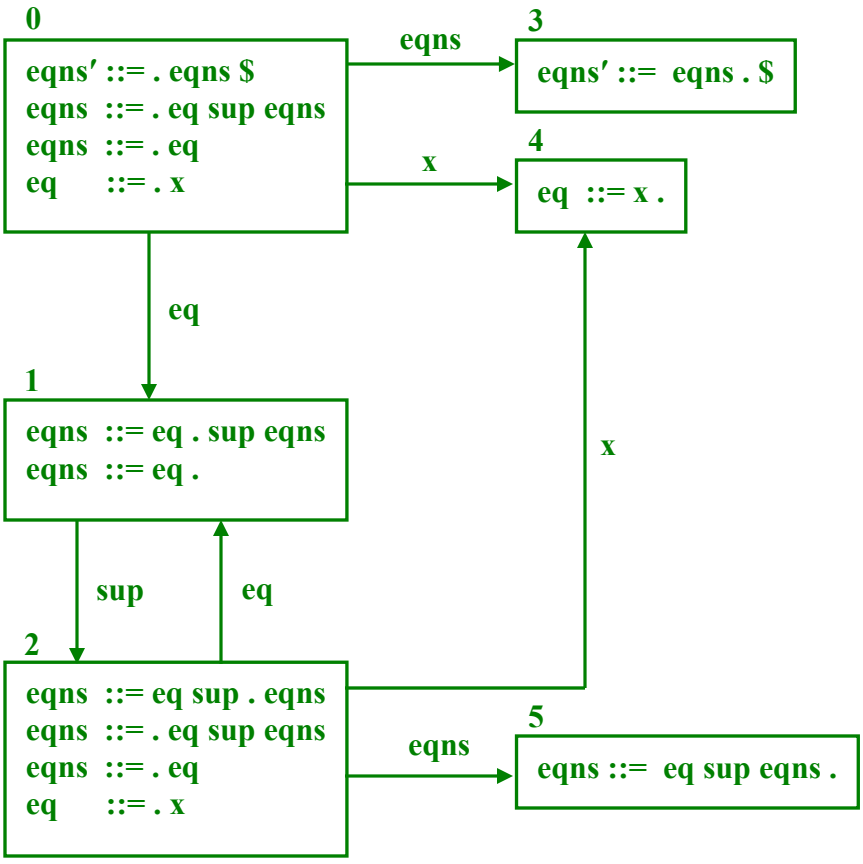
8. (20 points) (the inevitable LR parsing question) One of the first, and longest-lasting applications on the Unix©®™ operating system is text processing, particularly for documents containing mathematics. Unix includes a preprocessor eqn that translated equations written in a small language into appropriate typesetting codes. The eqn source language can be described by a context-free grammar, and lex and yacc can be used to implement the eqn translator.

A fragment of that grammar is the syntax for expressions involving superscripts:

0. *eqns*' ::= *eqns* $ ($ is the end-of-file marker)
1. *eqns* ::= *eq* sup *eqns*
2. *eqns* ::= *eq*
3. *eq* ::= x

For example, x sup x sup x describes the equation x^(x^x). (Note that, as in mathematics, multiple superscripts group to the right.) The symbols sup and x are terminals representing themselves

(a) (8 points) Construct a LR(0) state machine for this grammar. Be sure to show the set of items in each state.



(continued next page)

8 (cont.) Grammar repeated below for convenience

0. *eqns*' ::= *eqns* $ ($ is the end-of-file marker)
1. *eqns* ::= *eq* sup *eqns*
2. *eqns* ::= *eq*
3. *eq* ::= x

(b) (2 points) Is this grammar LR(0)? Why or why not?

No. There is a shift/reduce conflict in state 1 (shift on sup vs. reduce eqns ::= eq .).

(c) (4 points) Compute First, Follow, and Nullable for each of the non-terminals in the grammar.

	First	Follow	Nullable
eqns'	x		no
eqns	x	$	no
eq	x	sup, $	no

(d) (4 points) Write down the SLR(1) *action* and *goto* table for this grammar, using the information in the LR(0) diagram from part (a) and the First, Follow, and Nullable information from part (c) of the question.

state	x	sup	$	eqns	eq
0	s4			g3	g1
1		s2	r2		
2	s4			g5	g1
3			acc		
4		r3	r3		
5			r1		

(e) (2 points) Is this grammar SLR(1)? Why or why not?

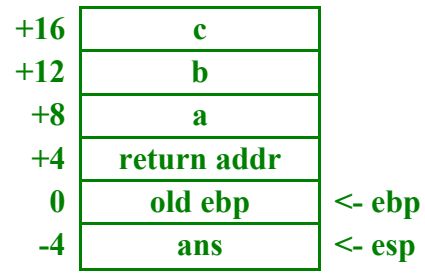
Yes. The 1-symbol lookahead information gets rid of the shift-reduce conflict in state 1, and there are no other shift-reduce or reduce-reduce conflicts.

9. (16 points) Consider the following integer valued C function:

```
int fun(int a, int b, int c) {
   int ans;
   ans = max(a, c);
   ans = max(ans, b);
   if (ans < 0) {
      ans = - ans;
   }
   return ans;
}
```

(a) (4 points) Draw a picture showing the layout of the stack frame after the function prologue has executed but before the first statement in the function body is executed. Be sure to show

- the locations and numeric offsets of all of the local variables and parameters, as well as any other information in this function's stack frame.
- where the `esp` and `ebp` registers point in the stack frame.



(b) (12 points) On the next page, write an x86 assembly-language version of this function.

- Your answer doesn't need to strictly imitate the code shape discussed in class – straightforward x86 code that reflects the source code is all that's needed, but…
- You need to use the proper C language calling conventions.
- Your assembly language code must include all of the statements shown, i.e., your code must show all the stores and loads implied by the various statements, even though this might not be strictly needed to achieve the same effect.
- Your code should address variables and parameters using the proper base register and offset in the stack frame.
- Assume that `max` is an integer-valued external function that you can call without requiring any further declarations.
- You do not need to include things like `.386`, `.model`, or other assembly language directives.

9. (cont). Function repeated for reference. Write your x86 assembly language version below. You just need to translate this function – don't worry about the code that calls it.

```
int fun(int a, int b, int c) {
   int ans;
   ans = max(a, c);
   ans = max(ans, b);
   if (ans < 0) {
      ans = - ans;
   }
   return ans;
}
```

This answer shows code using a single register as an accumulator. More compact code is possible, and as long as your answer included the addressing details (particularly the correct offsets from ebp for the different variables), it was fine to omit some redundant loads (a few of the most obvious are marked with *, below).

```
fun:  push ebp           ; prologue
      mov  ebp,esp
      sub  esp,4

      mov  eax,[ebp+16]  ; ans = max(a,c);
      push eax           ;  (could use "push [ebp+16]" instead)
      mov  eax,[ebp+8]   ;  (etc.)
      push eax
      call max
      add  esp,8
      mov  [ebp-4],eax

      mov  eax,[ebp+12]  ; ans = max(ans,b);
      push eax
      mov  eax,[ebp-4]
      push eax
      call max
      add  esp,8
      mov  [ebp-4],eax

      mov  eax,[ebp-4]   ; if (ans < 0)        *
      cmp  eax,0
      jnl  skip

      mov  eax,[ebp-4]   ;   ans = - ans;      *
      neg  eax
      mov  [ebp-4],eax

skip: mov  eax,[ebp-4]   ; return ans;         *
      mov  esp,ebp
      pop  ebp
      ret
```