Compilers Comprehensive, November 7, 2007

1. (10 points)

   Suppose a lexical analyzer generator such as lex, flex, jlex, or jflex, were given the series of patterns:

   ab
   abc
   (ab?c)*
   . (“.” matches any individual character except newline)

   What strings would be recognized in the input `abcaccaabbacac` by repeated calls to the lexer?

   Briefly explain why the lexer would do this.

2. (20 points)

   The following is the relevant fragment of a YACC (actually, Bison) grammar for simple Boolean formulas. This (admittedly odd) grammar translates Boolean expressions to “reverse Polish” notation: `a | b & c` is translated to “a b OR c AND.” But, it also transforms the expression to negation normal form, using De Morgan’s laws to “push NOTs down to the leaves of the formula,” and it does this *on-the-fly without using any additional data structures or variables.* The negation normal form of `~(a | ~(b & c) | ~d)` is `~a & b & c & d)`.

```
%union yystacktype {char *name;  int flag; }

%start Formula

%token <name> ID
%left '|'
%left '&'
%nonassoc '~'

%%

F : F '&' { $<flag>$ = $<flag>0; } F
    { if ($<flag>0) {printf("OR\n"); } else { printf("AND\n"); }}
  | F '|' { $<flag>$ = $<flag>0; } F
    { if ($<flag>0) {printf("AND\n"); } else { printf("OR\n"); }}
  | '~' { $<flag>$ = !$<flag>0; } F
  | ID { printf("%s\n", $1); if ($<flag>0) { printf("NOT\n"); }}
  | '(' { $<flag>$ = $<flag>0; } F ')'
  ;
```

   (a) If the `%left` and `%nonassoc` declarations are omitted, bison reports “6 shift/reduce conflicts.”

      i. What is a shift/reduce conflict in an LALR parser?

      ii. Describe one specific conflict that occurs in this example.

iii. The YACC input above does not produce any errors about conflicts, because they are all resolved. Explain why this happens and how it works.

(b) This grammar has actions embedded in the middle of the productions. In YACC-style parser generators, this is "syntactic sugar" for a grammar with more rules, but no embedded actions (actions appear only at the right-hand ends of productions, to be executed when those productions are reduced). Please explain this how to de-sugar a grammar like this into one where all actions occur when productions are reduced.

(c) Basically, how does the YACC grammar with actions work? How does the parser keep track of whether the next sub-expression is in a negated context, especially when it exits the scope of a negation (as when it finishes parsing `~(b & c)` in the above example)?

3. (15 points)

   Describe an implementation for a symbol table supporting the following operations. You should assume that, for practical purposes, hash table insertions and lookups require constant time. If your solution does not meet the performance goals below, it will still receive some credit so long as it is clearly correct. You may assume that a symbol is bound only once in each scope.

   - `pushscope()` – mark a new declaration scope, e.g., at the beginning of a function declaration. This should be a constant-time operation.
   - `popscope()` – restore the symbol table to its state just before the matching pushscope. This should take $O(k)$ time, where k is the number of symbols declared in the scope that is popped.
   - `declare(symbol, decl)` – Associate "symbol" with "decl" in the current scope. decl is a datastructure describing, for example, a variable or a type declaration for the symbol. This should take constant time (not counting the time to construct the decl).
   - `lookup(symbol)` – Find and return the decl most recently associated with symbol in a scope that has not been popped. This should take constant time.

4. (15 points) How does compile-time function (or method) overloading interact with type-checking in programming language implementations? Briefly discuss how type evaluation could work in two scenarios: when overloaded functions are resolved based only on the types of the arguments of the functions, and when the return type of overloaded functions is used to resolve overloading, in addition to the argument types. Don't worry about automatic type conversions (e.g., promotion from integers to floats) or inheritance.