Solutions to 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.

   Solution: `abcac|c|a|ab|b|acac`

   On each call, the lexer matches the longest string it can against its patterns. For example, `a`, `ab`, `abc`, etc. all match the first part of the input, but `abcac` is the longest matching prefix.

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?
Solution: It occurs in a parser state that have both shift and reduce items, where the lookahead symbols for the reduce item include the next terminal symbol in a shift item. E.g., the state could have items $[A \rightarrow \alpha \cdot a\beta, X]$ and $[B \rightarrow \gamma, a]$.
The parser must commit to a shift or reduce action without sufficient context to determine which will lead to a successful parse.
In this case, the problem is that the grammar is ambiguous, so both parses will be successful, but lead to different translations for the logical expression.

ii. Describe one specific conflict that occurs in this example. **Solution:**
Here are all of them:

```
F -> F & F . vs F -> F . & F. (left or right associative?)
F -> F | F . vs F -> F . | F. (left or right associative?)
F -> F | F . vs F -> F . & F. (| vs & precedence)
F -> F & F . vs F -> F . | F. (& vs | precedence)
F -> ˜ F . vs F -> F . | F.   (˜ vs | precedence)
F -> ˜ F . vs F -> F . & F.   (˜ vs & precedence)
```

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.
Solution: The `%left` and `%nonassoc` declarations define whether `&` and `|` are left or right associative, and the order of the declarations gives the relative precedence of the operators (lower precedence first). Bison/YACC/etc. resolve these conflicts by using the precedence rules to choose which action goes into the ACTION table entry.

(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.
Solution: For each action, a new non-terminal symbol is generated (say A_i and a new production is added (e.g., $A_i \rightarrow \epsilon$). The embedded action is performed when this new production is reduced. YACC goes to some special effort to make `$i` symbols in the action of this refer to the proper positions in the value stack, based on the positions of the corresponding symbols in the original rule, not the newly introduced rule.

(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)?
Solution: Whenever the grammar is about to parse a `F`, it makes sure that the value on top of the value stack is 1 iff `F` is in a negated context (it copies the flag as necessary). When parsing an `F`, this is `$0`, which refers to the position on the value stack just before `$1`, i.e. just under the position for the leftmost symbol of the RHS of the current production. When the parser reduces a `˜` formula, the flag saying whether the current context is negated, leaving the flag for the next outer context in the right position on the stack for the next formula.

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.

Solution: We assume that symbols are strings or records with strings an other data, and that decls are pointers to records representing decls. A symbol table entry is a record with a pointer to a symbol, a pointer to a decl, a "saved" link to another entry (or null) and a "next_in_scope" link to another decl. There is a hash table with symbols as keys and entries as values, and a separate "scopes" stack of pointers to entries.

`pushscope()`: pushes a null onto the scope stack.

`declare(symbol, decl)`: Allocate a new entry, set the symbol and decl fields to symbol and decl, set next_in_scope to the top entry in the scope stack, overwrite the top of the scope stack with the new entry, set the ''saved" field to the current value of key in the hash table (or null if there is no value), store key → symbol in the hash table.

`lookup(symbol) -> decl`: Lookup symbol in hash table. Report an error if there is no value. Otherwise, return the contents of the decl field in the entry that is found.

`popscope()`: Get entry on top of scope stack. Iterate over the linked list of entries created by the "next_in_scope" fields, removing each entry from the hash table using the "symbol" field and storing the "saved" value of the entry under the key (if non-null) Pop the scope stack. (Note: A nice implementation of a hash table might provide a way to update or delete the entry directly without hashing multiple times).

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.

   Solution: Without overloading, type checking usually involves assigning types to expressions bottom-up: given the types of the children of an operator or function call, the result type can be determined. Type errors are detected when the children of the operator or function have inappropriate types.

   If function overloading depends only on the function arguments (children), types can still be assigned by a bottom-up pass. Once the types of the arguments to a function call have been determined, the compiler can find the correct overloaded function definition that matches those types. If no such function exists, it is a type error. If there are multiple matching functions, it should also be an error. Otherwise, there is a unique function, and the return type of that function is the type of the current expression.

If the function return type can be used to resolve ambiguities, the match of the correct function depends on what type the surrounding context "wants." E.g., the function call is an argument to another function call, which needs a particular type, or is assigned to a variable of a particular type. If only one of the overloaded function definitions results in a correctly typed expression, that is the correct function to choose. If there are no functions or multiple functions, it is an error. In this case, assigning types involves both top-down and bottom-up passes – and may need to be repeated to propagate type constraints over the abstract syntax tree.