Solutions to Compilers Comprehensive, November 2005

This is a 60 minute, closed book exam. Please mark your answers in the blue book.

1. (10 points)

   Suppose you were implementing a lexical analyzer for the C programming language in a tool like Lex or Flex. Suppose the entire input to the compiler were:

   ```
   divbypointer(double num, double *pdenom)
   {
     return num/*pdenom;
   }
   ```

   Describe two ways of handling comments.

   (a) Method 1 uses a regular expression that matches a complete comment as a single lexeme.

   (b) Method 2 recognizes `/*` and then enters a special "start condition," in which `*/` and any single character are recognized as lexemes. When `*/` is recognized, it returns to the default start condition, in which C language tokens are recognized.

   Briefly discuss how each method would handle the example above, and discuss the practical merits of each.

   Solution:

   Method 1 would compile the above program without errors. There are many disadvantages to method 1. The regular expression is horrible; the lexer produces surprising results for unterminated errors in all but this useless example; and the lexer uses a lot of buffer space and time to keep track of the contents of comments, which are then thrown away. This is the wrong engineering approach.

   Method 2 would say the above program has an unterminated comment. Method 2 is easier and more efficient in time and buffer space. It also recognizes errors more reasonably. This is a much more efficient and easier approach.

2. (15 points) These questions are about the handling of variables by a compiler for a simple language like C. In your answers, address only the compiler behavior that is *necessary for code generation.* Do not address type checking or other aspects of semantic analysis are not strictly necessary to emit code for correct programs.

   (a) Describe the compiler's processing of a global variable g of type `int`, both at the point of *declaration* and at the point of *use*.

   Solution:

   At the point of declaration, the compiler needs to enter info g into the symbol table, including its type, etc. but, especially, its location. The location is a constant offset relative to a section of memory reserved for global variables. The offset is established when the variable is *allocated* within the global data section. The actual address of the global data section of memory is often not established until link time, but it is a constant when the program is actually executed.

At the point of use, the compiler needs to compute the address of the variable. If no array indexing is required (as when it is of type `int`, the address is a known constant, so no code needs to be generated. However, code is needed to fetch the value of the variable from the memory location.

(b) How is the handling of a local variable declaration of type `int` different from the handling of the global variable of the same type?

Solution:

Obviously, it is tagged as a local, not global variable when it goes in the symbol table. The offset is relative to a stack frame pointer of some kind, which is usually stored in a machine register. The stack frame pointer is set up when a function is called, and restored when the function returns.

When the variable is accessed, code is needed to add the contents of the frame pointer to the offset for the local variable. This is almost exactly the same code that would be required for an expression with $+$ in it. Once the location of the variable has been computed, code must be generated to fetch the value.

(c) What information does the compiler have to maintain in the symbol table to generate code for `A[i].f[j]`?

Solution:

It needs to keep track of whether the variable `A` is local or global, and what its offset is (as above). It also needs remember the sizes of the array elements (to do array indexing) and the offsets of fields within structures. The generated code computes (frame pointer) + (offset of A) + i * (size of A elements) + (offset of f) + j * (size of A[i].f elements)

3. (10 points)

Why would it be useful for an optimizing compiler to have optimizations on an intermediate representation (such as 3-address code) *and* peephole optimization at the instruction level?

Solution:

The most sophisticated optimizations occur on intermediate code, because

- more information about the programmer's intent is available at the intermediate code level.
- they are more-or-less machine-independent, so they don't have to be rewritten when the compiler is retargeted.
- the intermediate representation can be designed to make the optimizations easy to implement.

Even with very sophisticated optimizations on intermediate code, peephole optimization is still useful. Some optimizations depend on information that is not available until the instructions are generated. For example, some machines have long jumps and short jumps, and won't know whether a short jump can be used until it knows the layout of the instructions.

4. (35 points) Consider the following context-free grammar:

$$\begin{array}{lcl} S & \rightarrow & Sa \\ S & \rightarrow & bS \\ S & \rightarrow & c \end{array}$$

(a) (3 points) Show that the grammar is ambiguous.

Solution:

There are two leftmost derivations for "bca": $S \overset{L}{\Longrightarrow} Sa \overset{L}{\Longrightarrow} bSa \overset{L}{\Longrightarrow} bca$
$S \overset{L}{\Longrightarrow} bS \overset{L}{\Longrightarrow} bSa \overset{L}{\Longrightarrow} bca$

(b) (10 points) Write the canonical collections of LR(1) items for this grammar.

Solution:

$$\boxed{\begin{array}{lcl} S' & \rightarrow & \bullet S, \$ \\ S & \rightarrow & \bullet Sa, \$a \\ S & \rightarrow & \bullet bS, \$a \\ S & \rightarrow & \bullet c, \$a \end{array}} \quad \boxed{\begin{array}{lcl} S' & \rightarrow & S\bullet, \$ \\ S & \rightarrow & S \bullet a, \$a \end{array}} \quad \boxed{\begin{array}{lcl} S & \rightarrow & b \bullet S, \$a \\ S & \rightarrow & \bullet Sa, \$a \\ S & \rightarrow & \bullet bS, \$a \\ S & \rightarrow & \bullet c, \$a \end{array}}$$

$$\boxed{\begin{array}{lcl} S & \rightarrow & c\bullet, \$a \end{array}} \quad \boxed{\begin{array}{lcl} S & \rightarrow & Sa\bullet, \$a \end{array}} \quad \boxed{\begin{array}{lcl} S & \rightarrow & bS\bullet, \$a \\ S & \rightarrow & S \bullet a, \$a \end{array}}$$

(c) (2 points) Identify all conflicting items, and the types of the conflicts (e.g., "shift-reduce conflict in state 3 on d").

Solution: There is a shift/reduce conflict on a in this state:

$$\begin{array}{lcl} S & \rightarrow & bS\bullet, \$a \\ S & \rightarrow & S \bullet a, \$a \end{array}$$

This stems from the ambiguity in the grammar. E.g., should bca be parsed as $(bc)a$ (reduce at this state) or $b(ca)$ (shift the a).

(d) (5 points) Could the original grammar be converted into an LALR(1) parser that parses all input correctly by resolving conflicts, in the way that YACC and similar parser generators allow? If so, how should they be resolved? In either case, please explain (briefly).

Solution:

Yes. Surprisingly, we can always shift in the situation of the previous problem or always reduce. In either case, the correct language will be accepted, although different trees will result from the two different ways of resolving the conflict.

(e) (5 points) Rewrite the grammar in an equivalent form that is suitable for LL parsing and minimizes the use of stack space.

Solution:

$$\begin{array}{lcl} S & \rightarrow & BcA \\ B & \rightarrow & bB|\epsilon \\ A & \rightarrow & aA|\epsilon \end{array}$$

(f) (5 points) Rewrite the grammar in an equivalent form that is directly suitable for LR parsing (i.e., does not result in conflicts) and minimizes the use of stack space.

Solution:

$$\begin{array}{rcl} S & \rightarrow & BcA \\ B & \rightarrow & Bb|\epsilon \\ A & \rightarrow & Aa|\epsilon \end{array}$$

(g) (5 points) In your modified LL(1) grammar, show the sequence of stack contents and inputs when parsing the input $bbcaa$.

Solution:

(top) stack	parse	action
S$	bbcaa$	expand $S \rightarrow BcA$
BcA$	bbcaa$	expand $B \rightarrow bB$
bBcA$	bbcaa$	match
BcA$	bcaa$	expand $B \rightarrow bB$
bBcA$	bcaa$	match
BcA$	caa$	expand $B \rightarrow \epsilon$
cA$	caa$	match
A$	aa$	expand $A \rightarrow aA$
aA$	aa$	match
A$	a$	expand $A \rightarrow aA$
aA$	a$	match
A$	$	expand $A \rightarrow \epsilon$
$	$	accept