Compilers Solutions

There are a number of essay questions on this exam. You should, of course, keep your answers clear and concise.

1. (5 minutes) Answer true or false. In all questions, assume that the context-free grammar has no useless symbols.

   (a) There exists a context-free grammar with that is LL(1) but not LR(1).
   false

   (b) There exists an context-free language that is LR(1) but not LL(1).
   true, e.g. $\{a^i b^j \mid i > j\}$

   (c) YACC can correctly parse all ambiguous context-free grammars by using precedence and associativity hints.
   false, e.g. $\{a^i b^j c^k \mid i = j \text{ or } j = k\}$

   (d) There exists an unambiguous grammar that is not LR(1).
   true, e.g. $S \rightarrow Aaa|Bab; A \rightarrow \epsilon; B \rightarrow \epsilon$

   (e) LALR(1) parser generation (*not parsing*) is linear in the size of the input grammar.
   false - it's exponential in the worst case. Parsing is linear in the input size, though.

2. (10 minutes) Consider the following grammar:

$$
\begin{aligned}
S &\rightarrow AB \\
A &\rightarrow aA \\
A &\rightarrow \epsilon \\
B &\rightarrow b \\
B &\rightarrow SC \\
C &\rightarrow c
\end{aligned}
$$

   (a) Compute the FIRST and FOLLOW sets for the nonterminals of the grammar.

   First and Follow sets

Symbol	First	Follow
S	a, b	$c, \$$
A	ϵ, a	a, b
B	a, b	$c, \$$
C	c	$c, \$$

   (b) Based on the FIRST and FOLLOW sets, is it LL(1)? Explain your answer.

   The grammar is not LL(1), because the right-hand-sides of productions $B \rightarrow b$ *and* $B \rightarrow SC$ *both have* b *in their FIRST sets. Also, the* a *is in FIRST of* $A \rightarrow aA$ *and FOLLOW of* A *(relevant because of* $A \rightarrow \epsilon$ *production).*

   (c) What is the language of this grammar? From inspecting the language, is it obvious whether there exists an LL(1) answer (if you say it's obvious, explain [obviously]).

   The language of the grammar is regular (a^*bc^*)*, so there is an LL(1) grammar.*

(19)

3. (10 minutes) 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.*

   There is a large amount of latitude in the answers, depending on what assumptions one makes about the language and compiler. Here is an example:
   At the point of declaration, the compiler needs to enter info about g into the symbol table, including its type, etc. but especially, its location. The offset is established when the variable is allocated *within the global data section. The offset is a compile-time constant. 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?

   Obviously, the variable 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. Usually, all the locals for the procedure are allocated in a single stack frame for the procedure. 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]`?

   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)*

4. (5 minutes) Imagine a language in which the types of expressions can be determined in a single bottom-up traversal of the syntax tree of a program. The language also has implicit type conversions (also called *coercions*), such as `int` to `float`.

   How could a compiler

   - recognize when coercions need to occur, and
   - generate code to perform them

   in a single bottom-up traversal of an expression?

20

A coercion is necessary when there is a type mismatch between operands and operator, which can be detected during a bottom-up pass when the types of the operands have been determined and the operator is examined. For example, in the expression $i+x$, if i is an integer and x is a float, there is a type mismatch, because $+$ requires two integers or two floats.

Machines that support floating point have conversion instructions. The compiler acts as though it had inserted a conversion operation into the syntax tree at the point of the type mismatch, which it then compiles just like any other operator. For example, $i + x$ would become $flt(i) + x$ (which provides two float operands to $+$), and the appropriate machine instructions would be generated to compute the result of $flt(i)$, which is then added to x.

21