SOLUTIONS: Compilers

CS Comprehensive Exam: Compilers (30 points)

1. (7 points)

   Comments in the C programming language are introduced by the characters `/*` and terminated by the characters `*/`. They do not nest and they do not occur within string or character literals. Indicate whether or not each of the following regular expressions describes precisely the set of all valid C comments. For those that do not, provide a counterexample. Assume for convenience that the character set is {`*`,`/`,`a`}.

   (a) `/*(/|a|*)**/`
   Ans: no, this expression matches `/**/*/`.

   (b) `/*(a|/)**(*|a(a|/)**)*/`
   Ans: yes, perhaps most easily seen by constructing a DFA.

   (c) `/*(a|/)**(*|a(a|/)**)**/`
   Ans: no, this expression fails to match `/**/`.

2. (8 points)

   The following grammar generates all regular expressions over the alphabet $\{a, b\}$:

$$R \rightarrow R + R \mid RR \mid R^* \mid (R) \mid a \mid b$$

   where '+' denotes 'or'. Unary $*$ has highest precedence, followed by concatenation; disjunction has the lowest precedence. All operators are left associative.

   (a) Show that the grammar is ambiguous.
   Two leftmost derivations of aba are:

$$R \Longrightarrow RR \Longrightarrow RRR \Longrightarrow aRR \Longrightarrow abR \Longrightarrow aba$$
$$R \Longrightarrow RR \Longrightarrow aR \Longrightarrow aRR \Longrightarrow abR \Longrightarrow aba$$

   (b) Construct an equivalent unambiguous grammar that enforces the correct operator precedence and associativity.

$$\begin{aligned} R &\rightarrow R+T \\ R &\rightarrow T \\ T &\rightarrow TF \\ T &\rightarrow F \\ F &\rightarrow F^* \\ F &\rightarrow (E) \\ F &\rightarrow a \\ F &\rightarrow b \end{aligned}$$

(c) Eliminate any left-recursive productions and productions with common prefixes from your grammar in (b).

$$\begin{array}{lcl}
R & \rightarrow & TA \\
A & \rightarrow & +TA \\
A & \rightarrow & \epsilon \\
T & \rightarrow & FB \\
B & \rightarrow & FB \\
B & \rightarrow & \epsilon \\
F & \rightarrow & CD \\
C & \rightarrow & (E) \\
C & \rightarrow & a \\
C & \rightarrow & b \\
D & \rightarrow & {}^*D \\
D & \rightarrow & \epsilon
\end{array}$$

3. (4 points) What is meant by ***structural equivalence*** and ***name equivalence*** of types? Is one more restrictive than the other? Is one easier for a compiler to check than the other?

Two type expressions are structurally equivalent if they are the same basic type (*e.g.*, integer or character), or if they are formed by applying the *same* constructor to structurally equivalent types (*e.g.*, *pointer*(T_1), *pointer*(T_2), where T_1 and T_2 are structurally equivalent types). When type expressions can be named, two type expressions are name equivalent precisely when they are identical. Name equivalence is more restrictive, and implies structural equivalence. Two types may be structurally equivalent but not name equivalent.

Checking expressions for name equivalence is particularly simple. Structural equivalence is defined recursively, so a compiler may check the types recursively. Alternatively, a compound type may be encoded explicitly as a type attribute to facilitate equivalence checking. In general, checking structural equivalence requires more work.

4. (11 points)

Suppose you are asked to extend the C programming language to allow overloading of user-defined functions, based on the number of formal parameters and their types.

Answer the following questions as briefly, ***yet clearly***, as you can. We are more interested in abstract operations (*e.g.*, "store x in a table, along with its size") than in implementation details (*e.g.*, "for performance, implement the table with a doubly-linked list of AVL trees").

(a) How can a compiler keep track of the parameters and their types? Describe what should be recorded when a function declaration is processed.

We'll assume that types are represented in the compiler as pointers to structures, called `decls`, where each `decl` represents a type. We'll also assume that names are associated with types, variables, *etc.*, using a scoped symbol table. A symbol table entry for a user-defined function will bind a function name to a *list* of its function `decls`.

When the compiler begins processing a function declaration, it creates a `decl` to represent the function type that contains the function name, return type, the names and types of its formal parameters, and other relevant information. The compiler creates a new scope and declares the formals in it, then processes the function body in yet another new enclosed scope. After processing the function body, the 'formals' scope can be saved in the `decl`, say as a list of (name, type) pairs. The function `decl` is then added to the `decl` list in the symbol table entry for the function's name.

(b) How can a compiler determine which function to invoke in a procedure call? Describe what and how information can be retrieved when resolving a procedure call.

The compiler must determine the number and types of the actuals, and compare them with the formals lists stored in the symbol table in the list of function `decls`. A simple-minded way to do this is to construct a list of actuals types and then look for a match with one of the formals lists stored in the function's type `decls` in the symbol table.

If a match is found, the compiler should check that the matched function declaration has a valid return type for the call. Otherwise the compiler should issue a type error message.

(c) Describe type errors that can occur as a result of this language extension, and how a compiler can detect them.

A couple of type errors that can arise are unresolvable function declarations (*e.g.*, C uses structural equivalence for all types except structures, unions, and enumerations, so structurally equivalent formals are not differentiable), and a function call that does not match any function declaration, either because the number of arguments does not match, or the types do not match.

Before adding a function type `decl` to the `decl` list in a symbol table entry for the function name, the compiler should verify that the declaration is resolvable by the formals list. This can be accomplished by comparing the saved formals list in the function `decl` with the formals lists of all previously declared functions with the same name. If two formals lists cannot be resolved, the compiler should issue a type error message.