CS 434 Meeting 6 — 2/21/06

Announcements

1. When you are done with phase 1.1, simply type "make submit".

Type Checking

1. Type checking is basically the enforcement of a set of rules designed to ensure that only valid operands are provided to the various operations that can be invoked within a program.

2. To make the enforcement of such rules possible within a compiler, the compiler must have an encoding that can be used to represent the types associated with expressions by the rules of the type system.

   - An obvious approach to handling such type specifications, is to provide distinct descriptors (i.e. structures or objects) for type specifications.
     - Pointers to type descriptors would be found in:
       (a) the "variable type" fields in declaration descriptors for variable names,
       (b) return type fields,
       (c) the "type" fields in formal parameter name descriptors, and

3. Structured types are not really "described" by any single structure. Instead, the descriptors for structured types act as the roots of trees of descriptors that represent the full structure of the type.

   - The tree structure must be traversed to determine the type of a complex "variable" like `x.b(3).c`.

4. One aspect of the type rules of a language that determines the degree to which the trees/graphs representing types must be traversed is the language's definition of type compatibility or "matching"

5. No matter how type equivalence is defined, there are usually lots of things to check.

   - The types used on the left and right hand sides of assignment statements must "match".
   - Method calls must be checked both to make sure that the numbers of actuals and parameters match and that their types match.
     - Warning. The ordering of parameter descriptors can cause annoying problems.
   - You must check the operand types for each built-in operator.
     - In general, processing operators can be complicated by implicit conversion rules (e.g. integers become reals when added to reals).
     - Checking most binary and unary operators is easy in Woolite. They all take and return integers. The equality operator is an exception since you can use it to compare values of other types
   - Your routine to process variables should check:
     - That names used as variables are variables!
     - That the type of the sub-variable is some array type when processing an Nsubs.
   - The types required by expressions used in certain contexts are fixed by the language.
     - Most languages require booleans as conditions in ifs and whiles (Woolite requires integers).

6. Semantic processing routines often return values that summarize the properties of the sub-phrases to which they were applied.

   - For example, to make type checking possible your function to process expressions should return the type descriptor for the type of the expression.

Handling Errors in Declarations

1. In general, you want to avoid generating multiple error messages in response to a single mistake in the program.

2. It is quite easy for an error in a declaration to produce a flood of error messages.

   - Consider what happens if you accidentally declare something to be of the wrong type or mis-spell a variable' s name in it declaration. Every reference to the variable might produce a type error.
   - If you naively follow the rule that an expression is a type error unless both of its operands have appropriate types, then a expression that contains an error will cause a redundant error in every level of the larger expression in which it appears

3. There are several ways that you can minimize the number of redundant error message you generate.

- If an error is detected in a declaration, attempt to build an (incomplete) declaration descriptor anyway.
  - This can avoid a flood of "undeclared variable" messages later.
- If you can not associate a type with an object, set the type field in its descriptor to a value indicating that an error was detected when attempting to resolve the type. Then, don't generate an error when you find yourself processing an expression one of whose operands is of this "error" type (NULL may make a good error value).
- Return the "unknown type" value when you process any expression whose type can not be determined due to an error.
- Create a dummy declaration for any reference to an undefined variable once you have produced at least one message announcing the problem so that later references will not result in errors. To do this right, such dummy declarations may need to be *carefully* exported when you exit a scope (you don't need to do this).

Informal Formal Grammars

1. Basic introduction to context-free grammars.

   - A grammar is a specification for a language that is composed of rules like the following which (informally) says that anything composed of an expression followed by an assignment operator and a variable is a valid statement.

     $<$ stmt $> \rightarrow <$ var $> = <$ expr $>$

     This symbols in angle brackets denote classes of syntactic phrases.
   - The syntactic phrases in most interesting grammars are frequently defined recursively (either directly or indirectly).

     $<$ stmt $> \rightarrow$ **while** ($<$ expr $>$) $<$ stmt $>$
   - When used as a notation for specifying languages, various notational conveniences are employed (such as using a | to abbreviate a set of rules that would have the same phrase type on the left hand side).

     $<$ stmt $> \rightarrow <$ var $> = <$ expr $>$
     | **while** ($<$ expr $>$) $<$ stmt $>$

Formal Grammars

1. Context free grammars are a notation for describing sets of strings (each phrase type is really just the name of a set of strings). So, we start our formal study of grammars with some basic definitions concerning strings and sets of strings:

   Alphabet A finite, non-empty set of symbols.

   String A string over some alphabet (Σ) is a finite, possibly empty sequence of symbols from Σ. We will use ϵ to denote the empty string.

   Language A language over an alphabet is just a set of strings over that alphabet.

   Concatenation If x and y are two strings over Σ, then their concatenation, xy, is the sequence of characters obtained by placing the sequence of characters in x before the sequence y. If z = xy is a string, we say that x is a *prefix* of z and y is a *suffix* of z.

   Products If X and Y are languages over some alphabet, then their product, XY, is defined to be:

   $$\{xy \mid x \in X and \;\; y \in y\}$$

   Powers If X is a language over Σ we define X^n to be the language containing only the empty string if n = 0 and XX^{n-1} otherwise.

   Closures If X is a language over Σ we define X^+, the positive closure of X, to be the union of the sets X^1, X^2, X^3, . . . and X^*, the closure of X, to be the union of X^0 and X^+.

2. Now, the definition you have all been waiting for:

   Context-free Grammar A context free grammar is composed of:

   (a) A finite alphabet V_t called the terminal symbols.

   (b) A finite alphabet V_n called the non-terminal symbols.

   (c) A distinguished element of the set V_n denoted by the symbol S and referred to as the goal symbol or start symbol.

   (d) A finite set P of pairs composed of one element from V_n and one element from $(V_t \cup V_n)^*$ called productions. Productions are written in the form:

   $$A \rightarrow X_1 X_2 ... X_m$$

3. There is an interesting, alternate approach to interpreting the productions of a grammar. Rather than viewing them as rules for producing strings that belong to the language defined, we can view them as set inequalities over a set of variables composed of the non-terminal symbols.

   - In this interpretation,

     $<$ stmt $> \rightarrow$ **while** $<$ expr $>$ **do** $<$ stmt $>$

     is interpreted as

$<$ stmt $>$ $\supseteq$ **while** $<$ expr $>$ **do** $<$ stmt $>$

where $<$ stmt $>$ and $<$ expr $>$ are viewed as the names of sets of strings (i.e. sub-languages) and keywords and other terminal symbols denote the set containing just that symbol (i.e. **do** denotes { "do" }).

- The "interesting" thing about this view is that there are multiple solutions to the set of inequalities corresponding to the typical set of productions. In particular, letting all non-terminals denote Σ^* generally does the trick. The " standard " interpretation of productions corresponds to the smallest sets that satisfy the inequalities interpretation.

  The "standard" interpretation is based on the next concept we consider, the derivation.

4. The association between a context free grammar and the language it describes is formalized through the notion of a derivation:

   Direct derivation Given a grammar, G = (V_t, V_n, S, P), and two strings x and y in $(V_t \cup V_n)^*$ such that $x = \alpha A \beta$ and $y = \alpha\gamma\beta$ where $\alpha, \gamma, \beta \in (V_t \cup V_n)^*$ and $(A, \gamma) \in P$ we say that x *directly derives* y. In this case we write

   $$x \Longrightarrow y$$

5. Examples of direct derivations.

   Consider G =

   $<$ blob $>$ $\rightarrow$ x $<$ glob $>$ $<$ blob $>$ y
   $<$ blob $>$ $\rightarrow$ z
   $<$ glob $>$ $\rightarrow$ a $<$ glob $>$
   $<$ glob $>$ $\rightarrow$ ϵ

   - $<$ blob $>$ $\Longrightarrow$ x $<$ glob $>$ $<$ blob $>$ y.
   - x $<$ glob $>$ a $\Longrightarrow$ x a $<$ glob $>$ a

6. More on the notion of a derivation:

   Derivation Given a grammar, G = (V_t, V_n, S, P), and two strings x and y in $(V_t \cup V_n)^*$ we say that x *derives* y if there exists a sequence of string $\alpha_0, \alpha_1, \alpha_2, ..., \alpha_m$ all in $(V_t \cup V_n)^*$ such that

   (a) for all $i < m$, $\alpha_i \Longrightarrow \alpha_{i+1}$,

   (b) $x = \alpha_0$, and

   (c) $y = \alpha_m$.

   In this case we write

   $$x \overset{*}{\Longrightarrow} y$$

   The sequence $\alpha_0, \alpha_1, \alpha_2, ..., \alpha_m$ is called a derivation of length m of y from x.

7. Using the grammar G shown above we can say that that $<$ blob $>$ $\overset{*}{\Longrightarrow}$ xaxzyy since:

   - $<$ blob $>$ $\Longrightarrow$ x $<$ glob $>$ $<$ blob $>$ y
   - x $<$ glob $>$ $<$ blob $>$ y $\Longrightarrow$ x a $<$ blob $>$ y
   - x a $<$ blob $>$ y $\Longrightarrow$ x a x $<$ glob $>$ $<$ blob $>$ y y
   - x a x $<$ glob $>$ $<$ blob $>$ y y $\Longrightarrow$ x a x $<$ blob $>$ y y
   - x a x $<$ blob $>$ y y $\Longrightarrow$ x a x z y y

8. Time for more definitions:

   Sentential form Given a grammar, G, a string is called a *sentential form* of G if it is derivable from the start symbol of G.

   Sentence A sentential form containing only symbols from the terminal vocabulary of a language is called a *sentence*.

   L(G) The language defined by a grammar G is the set of all sentences.

   $$L(G) = \{s \mid S \overset{*}{\Longrightarrow} s \ \&\ s \in V_t^*\}$$

Parse Trees and Ambiguity

1. For our purposes, it is not enough that a grammar specify the set of strings that belong to a language, we also want a grammar to impose grammatical structure on strings since this structure influences the meaning associated with a string.

2. It is not clear a derivation provides the information we want about grammatical structure since a single string may have many derivations.

   - We have seen that given the grammar:

     $<$ blob $>$ $\rightarrow$ x $<$ glob $>$ $<$ blob $>$ y
     $<$ blob $>$ $\rightarrow$ z
     $<$ glob $>$ $\rightarrow$ a $<$ glob $>$
     $<$ glob $>$ $\rightarrow$ ϵ

     we can say $<$ blob $>$ $\overset{*}{\Longrightarrow}$ xaxzyy since:

- $< \text{blob} > \implies \text{x} < \text{glob} > < \text{blob} > \text{y}$
- $\text{x} < \text{glob} > < \text{blob} > \text{y} \implies \text{x a} < \text{glob} > < \text{blob} > \text{y}$
- $\text{x a} < \text{glob} > < \text{blob} > \text{y} \implies \text{x a} < \text{blob} > \text{y}$
- $\text{x a} < \text{blob} > \text{y} \implies \text{x a x} < \text{glob} > < \text{blob} > \text{y y}$
- $\text{x a x} < \text{glob} > < \text{blob} > \text{y y} \implies \text{x a x} < \text{blob} > \text{y y}$
- $\text{x a x} < \text{blob} > \text{y y} \implies \text{x a x z y y}$

- It is also possible to show this using the derivation:
  - $< \text{blob} > \implies \text{x} < \text{glob} > < \text{blob} > \text{y}$
  - $\text{x} < \text{glob} > < \text{blob} > \text{y} \implies \text{x} < \text{glob} > \text{x} < \text{glob} > < \text{blob} > \text{y y}$
  - $\text{x} < \text{glob} > \text{x} < \text{glob} > < \text{blob} > \text{y y} \implies \text{x} < \text{glob} > \text{x} < \text{blob} > \text{y y}$
  - $\text{x} < \text{glob} > \text{x} < \text{blob} > \text{y y} \implies \text{x} < \text{glob} > \text{x z y y}$
  - $\text{x} < \text{glob} > \text{x z y y} \implies \text{x a} < \text{glob} > \text{x z y y}$
  - $\text{x a} < \text{glob} > \text{x z y y} \implies \text{x a x z y y}$

3. Representing structural information as a tree provides a better way to eliminate irrelevant choices about the order of non-terminal expansion.

   Parse Tree A *parse tree* for a sentential form S of a grammar G is a tree in which each node is labeled with an element of $(V_t \cup V_n)$ in such a way that:
   (a) The root is labeled with the start symbol,
   (b) the nodes of the frontier of the tree (i.e. the leaves of the tree) are labeled with the symbols that form S, and
   (c) each interior node is labeled with some non-terminal, N, such that $N \rightarrow \alpha \in P$ and α is the string of labels found on the node N's children.

4. A parse tree for our favorite string and grammar (the old xaxzyy example) is shown in Figure 1.

5. There are grammars in which certain sentential forms have more than one parse tree.

   - $< \text{stmt} > \rightarrow$ **if** $< \text{expr} >$ **then** $< \text{stmt} >$
     $\mid$ **if** $< \text{expr} >$ **then** $< \text{stmt} >$ **else** $< \text{stmt} >$

   Ambiguity A grammar G is said to be *ambiguous* if there is some string in L(G) with two distinct parse trees.

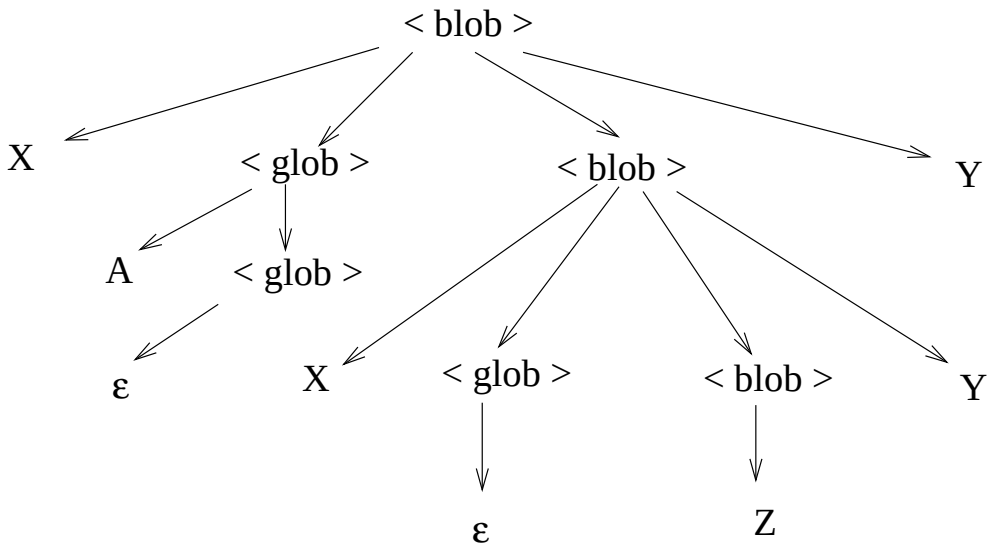

Figure 1: A parse tree for xaxzyy