CS 434 Meeting 1 — 2/2/06

Introduction

1. Attendance
2. Nature of Compiler Design
   - Lot's of "practice".
   - Lot's of "theory".
3. Nature of the Course
   - Learning how to write a bad compiler
   - Very project oriented
   - Majority of "theory" after in second half of semester
4. Handout Syllabus
   - Most current "syllabus" is course web page:
     - The class web page can be found at: http://www.cs.williams.edu/˜tom/courses/434
   - Discuss Text + nature of readings.
     - Students who are not comfortable in C should definitely get a C book and start learning. I recommend the Kernighan and Ritchie *C Reference Manual* as the quickest way for someone who already knows how to program to learn C.
     - I strongly suggest that you buy *Engineering a Complier* by Cooper and Torczon. It is definitely a good compiler text. I'm just not sure how closely I will be able to incorporate it (or any other) compiler text in the course.
   - The truth about office hours.
     - My door is always open, but
     - I won't always be there.
5. A note on class notes
   - I won't usually distribute copies, but the will be available on the course web page (sometimes even before class).
   - don't laugh, but do inform me of gross errors.
6. The Project
   - Implement:

     Woolite An object-oriented language including Java-like syntax but simplified in many ways (no interfaces, only one primitive type (int), only public, non-static methods, only private instance variables, no constructors, etc.).
     - This is the first time Woolite has been used as the target language, so it should be interesting!
   - Target machine will be the WC34000!
   - Need to know Unix and C.
   - Team projects strongly encouraged. Two is my favorite number.
   - Discuss workload implications of the project.
     - Requires a great deal of programming and it is cumulative.
     - Deadlines are somewhat flexible but lateness tends to be cumulative (and fatal).
     - Failure to produce a working compiler = failure.
7. Honor Code

The Function and Structure of a Compiler

1. The function of a compiler
   - Implementation of a higher level virtual machine through translation rather than interpretation.
     - It is worth noting that in this sense 434 has a natural connection to 237 and 432. Together all these courses contribute to explaining how what you type as a line in a Java program ultimately turns into electrons flowing through a transistor.
   - Various forms compiler output can take:
     - machine code
     - object file
     - assembly language
     - C code, Java code
     - encoded form for interpretation (Java virtual machine code)
   - The fuzzy line between compilers and interpreters
2. The structure of a compiler
   - The BIG Picture

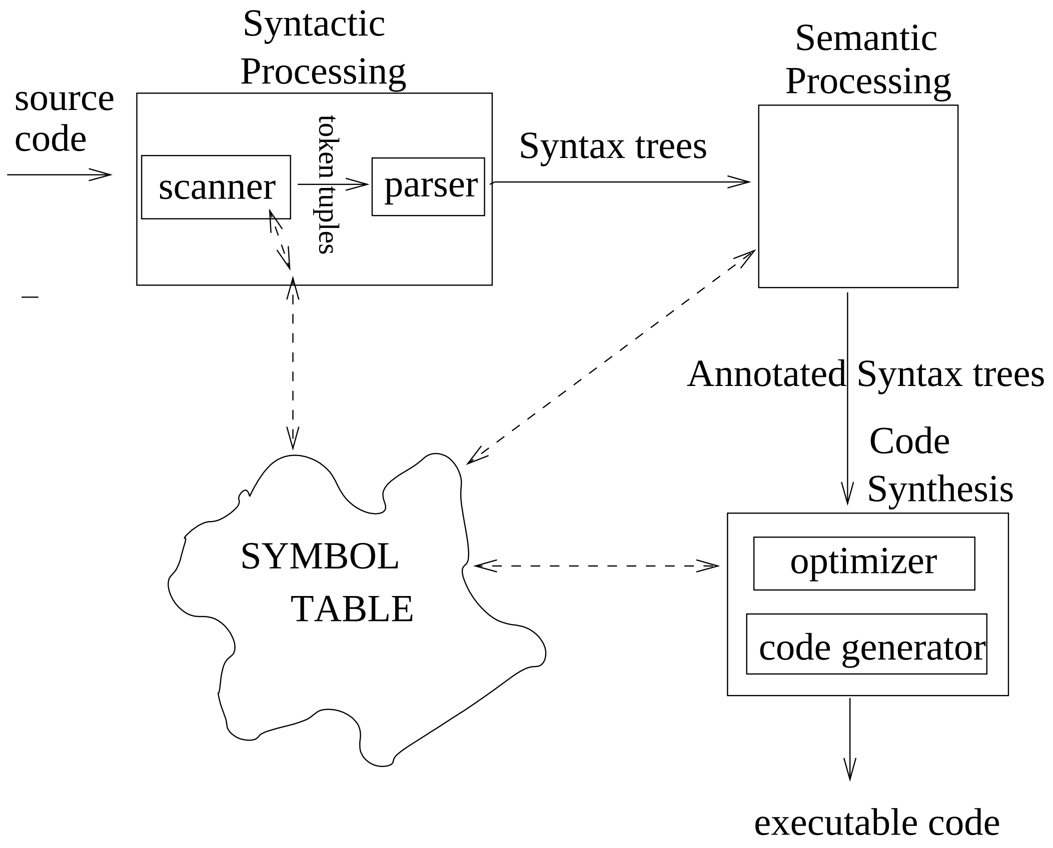

Figure 1: The BIG PICTURE

- General Roles of the Phases
  - Syntactic Analysis
    - ∗ Syntactic structure determines meaning.
      - · Consider importance of recognition of syntactic structure in natural language ("time flies like an arrow." vs. "Fruit flies like an orchard.")
      - · Consider an example from Java such as "Age[10]" which could either be part of a constructor (add "new") or a reference to an array element
      - · Consider the C code "(a) - b" which could be a strange way of writing a subtraction (if a and b are both of numeric types) or a cast of the result of the negation of the value of b to the type "a" (if a had been declared using "typedef a float;").
    - ∗ Syntactic analysis is the process of transforming a representation of a program as a sequence of characters into a representation in which the structure of program is explicit.
  - In most compilers, syntactic analysis is subdivided into two subphases:
    - ∗ Parsing: the process of grouping words and phrases into larger phrases of identifiable types.
    - ∗ Scanning: the process of grouping characters into "words". Scanning is also sometimes distinguished from syntactic analysis and called lexical analysis.

      It is hard to imagine that our brains don't have a similar separation that allows us to use the same mechanisms to parse a sentence whether we read it or hear it.
  - Semantic/Context sensitive Analysis
    - ∗ Observe that recognizing that the context in which the phrase 'Age[10]' appears requires an expressions does not imply we know the "meaning" of the phrase. In fact, if 'Age' is never declared or is a class name then the phrase is meaningless.
    - ∗ Semantic Analysis completes the internal representation of a program by processing declarations and verifying the legality of the program with regard to type and scoping rules.
    - ∗ Together, syntactic and semantic analysis enable the compiler to "understand" the source program.
  - Code Synthesis
    - ∗ Produces code from completed internal representation of the program.

    * Code synthesis may be done in many sub-phases depending on the quality of code desired.

- These "phases" of a compiler can be implemented in many ways.
  - In many compiler they are tightly integrated. The parser "method/procedure" invokes the scanner to get a token and then invokes the semantic analyzer or code generator to process each "phrase".
  - In our discussion, we will treat the phases of a compiler as separate processes sending data to one another.
  - It is probably best not to think of them as processing concurrently. Instead think of each phase as taking a complete representation of a program in one form and producing a new, complete representation in some other form.

Intermediate Representations

Now that we have an understanding of the basic phases of a compiler, we will discuss the types of data that flows between these phases. I will not attempt to describe every scheme for representing the intermediate results of the compilation process. I just want to show you one way to give you a concrete picture of what goes on. The organization I will outline corresponds closely to what will be used in the course project.

1. Source Code.
   - The scanner begins with the representation of the program with which we are all most familiar: A simple sequence of characters.
   - This representation is difficult to work with. If you have ever written a program that had to accept fairly free-format input you know the joys of skipping blanks and watching out for new-lines (not to mention tabs).
2. Tokens
   - A token is an independently meaningful subsequence of characters in a program. Identifiers, keywords, string constants and operators are all tokens.
   - The scanner converts the program from a sequence of characters into a sequence of tokens.
   - In the scanner's output, each token is typically represented by a pair of values. The first value indicates the token class (identifier, integer constant, etc.). The second value — the token identifier or token number — distinguishes tokens within the same class.
   - The use of a pair of values to represent tokens follows from the different interests of the phases that consume the scanner's output:
     - The parser doesn't need to know what identifier or constant it is looking at to determine the syntactic structure of some input. It just needs to know that it *is* looking at an identifier rather than a constant, a plus sign, or some keyword.
       The first element of a pair used to represent a token provides all the information the parser needs.
     - The semantic processor definitely needs to know what identifier or constant was used.
       The second element of a token pair encodes this information. The parser typically just passes this on to the later phases without examining it.
   - Thus, the scanner transforms the program from a sequence of characters to a sequence of pairs.
3. The Symbol Table
   - The scanner needs some set of values it can use as token identifier for program variables and method/function names.
   - The purpose of these token identifiers is to somehow let later phases of the compiler efficiently determine which source program name occurred in a given context.
   - Actually, what the later phases will really typically need to know is the attributes associated with the name that appeared in a given context (its type and memory location) rather than its actual identity.
   - This can be accomplished by keeping the attributes for each symbolic name in an object/structure and using a pointer/reference to this object as the token number.
   - This collection of objects is called the *symbol table*
4. Syntax Trees
   - There are many possible internal forms that can be produced by the syntax analyzer. We will use what are called *syntax trees.*
   - BEWARE: Syntax trees are similar to but different from *parse trees.* To emphasize the difference they are sometimes called *abstract syntax trees.*
   - The basic idea behind a syntax tree is a generalization of expressions trees. Consider the expression tree for the expression a*(b+c) shown in figure 2.

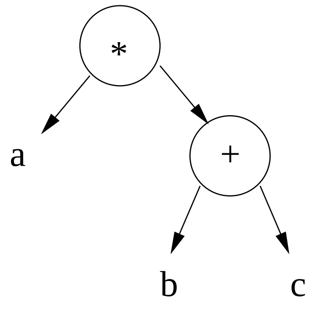

Figure 2: Expression Tree

  - Each subtree represents a phrase.
  - The root of the subtree is labeled with the "type" of the phrase.
  - The subtrees attached to an internal node represent sub-components of a phrase.

- In a general syntax tree, one simply recognizes that the root labels are really phrase types, not operators. For example, the syntax tree for:

  if (x <= 0) { flag = ; }

  would look something like the tree in figure 3.

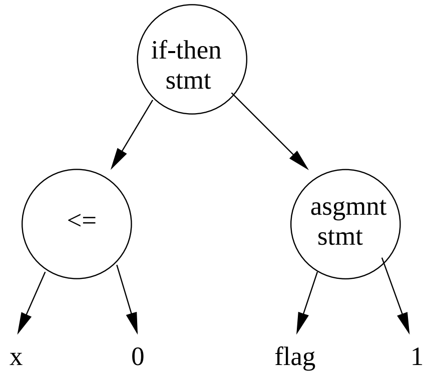

Figure 3: Syntax Tree for If Statement

- Note that:
  - In the character form of a program, keywords and punctation characters determine the phrase structure of a program. In the tree form, the internal nodes identify phrase types.
  - As a result, the only tokens that appear in such a tree are identifiers, constants and the like. Keywords and operators determine the form of the tree but they don't appear in it.
  - The tokens that do appear form the leaves.
- As a result, pointers from internal nodes of a syntax tree to its leaves are typically actually pointers to objects in the symbol table. (Note: This means the tree really isn't a tree).

5. Semantic Information

   - In your compilers, semantic processing will not produce a new representation of the program. It will simply "complete" the existing one.
     - For example, during semantic processing attributes of identifiers are extracted from declarations and stored in symbol table entries.
   - In the descriptions of other compilers found in the readings, you will find discussion of other intermediate forms including three address code, quadruples, etc.
     - In my view, these intermediate forms are more appropriately described when considering the sub-phases of code synthesis.

Block Structure and Symbol Table Organization

1. First, recall the rules of nested block structure.

   - A scope is a subsection of a program's text typically corresponding to a procedure/function/method/class or a block of statements.
   - An identifier can have only one definition in a given scope.
   - An identifier can be used in a scope as long as it is either defined in the scope or in a containing scope. (Note: Sometimes forward references are not allowed.)
   - Many programming languages allow arbitrary nesting of scopes.
     - Pascal functions/procedures
     - Java inner classes
     - ML let constructs
   - Any use of an identifier refers to the declarations of the identifier occurring in the smallest enclosing scope.

  – Textual nesting of scopes forms a tree.

2. For example, consider the interpretation of identifier references in the program whose skeleton is shown in Figure 4. The tree of scopes corresponding to this program text is shown in Figure 5.

```
class Program {
      int W;
      int X;
      class A {
           int W;
           int Y;
           int Z;

           void B() { int X ; . . . }
           void C() { int X; int Y ; . . . }
           . . .
      } // end of A

      void D() { int Z ; . . . }
}
```

Figure 4: A class definition skeleton illustrating nested declarations

3. The organization of many compiler symbol tables is fairly complex as a result of the need to support the scope rules associated with block structure. The standard approach to explaining symbol table organization, however, adds additional complexity by failing to properly distinguish the role of the scanner in building the symbol table from that of the semantic analysis routines in completing it.

4. Many compiler texts describe the symbol table as a dictionary, typically implemented using a hash table or a search tree.

   - In my (somewhat odd) view, the symbol table is simply a collection of objects/structures in which the attributes of identifiers are stored. The hash table (or whatever else is used) is simply a mechanism that enables the scanner to associate symbol table entries with the character string form of identifiers it processes.
   - Once the program has been transformed into a syntax tree, the semantic analyzer has direct access to symbol table entries through pointers stored in the tree. The hash table used by the scanner is not needed. Thus, it seems wrong to me to talk about the hash table as if it were the symbol table.

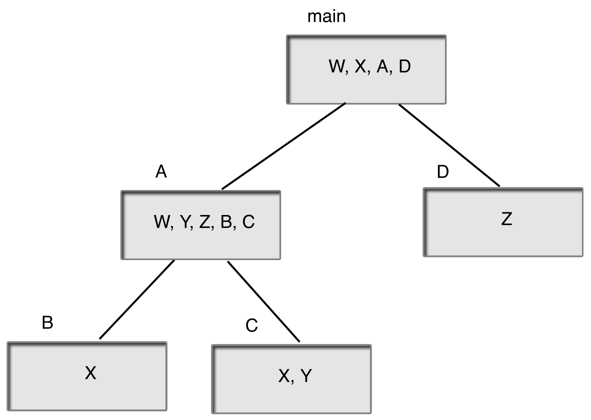

Figure 5: Tree of scopes corresponding to Figure 4

5. In the "symbol table is a hash table" view of a compiler, detecting undeclared and multiply declared identifiers becomes a task split between the scanner, parser and symbol table routines.

   - The symbol table interface provides (at least) two functions:
     (a) A function `find` which takes an identifier name and returns either a pointer to the associated symbol table entry or an error return code indicating that the identifier was not found.
     (b) A function `enter` which takes an identifier name and either creates a new entry for the identifier and returns a pointer to it or raises an error if an entry already exists.
   - The syntax analyzer has to somehow tell the scanner whether it is currently processing a declaration or a use. Things get even worse if block structure is involved.

6. A cleaner separation of function can be arranged. Only one search function is needed:

- A function `lookup` which takes an identifier, searches for an associated entry (creating a new one if necessary) and returns a pointer to the associated entry.

The errors that would have been raised by `find` and `enter` can instead be recognized during semantic analysis.

- Each symbol table entry would include a boolean '`defined`' attribute.
- When the semantic routines process a declaration, they would check to see if it had already been defined and raise an error if appropriate.
- Similarly, a use of an undefined identifier would cause an error.

7. The basic idea is to let the syntactic analyzer simply process all references to an identifier by mapping them to a pointer to the symbol table entry for the identifier. The semantic analysis phase then distinguishes declarations from uses by either putting information into or taking information out of the symbol table entry.