Lecture 10 Outline

- Semantic Analysis
- Symbol Tables
- Types

Reading:
Sections: 7.6, 4.8, 4.9, 5.1

Semantic Analysis

- Type checking
- Code generation/usually intermediate code
- Find syntax errors not caught by parser - match declaration and use - last source program errors found

Compiler phases & errors

- Lexical analysis
  - detects inputs with illegal tokens
- Parsing
  - detects inputs with incorrect structure
- Semantic analysis
  - all remaining syntax errors
  - last of the front end phases
Parsing cannot catch some errors

- Some language constructs are not context-free
  - Example: Identifier declaration and use
  - An abstract version of the problem is:
    \( \{ wcw \mid w \in (a \mid b)^* \} \)
  - The 1st \( w \) represents a declaration; the 2nd \( w \) represents a use

Semantic Analysis performs checks

- coolc checks:
  - All identifiers are declared
  - Types
  - Inheritance relationships
  - Classes defined only once
  - Methods in a class defined only once
  - Reserved identifiers are not misused
    And others . . .
  - The requirements depend on the language

Scope

- Matching identifier declarations with uses
  - Important static analysis step in most languages
  - Including COOL
  - Sometimes called static semantics
What's Wrong?

- Example 1
  Let y: String ← "abc" in y + 3

- Example 2
  Let y: Int in x + 3

Scope (Cont.)

- The scope of an identifier is the portion of a program in which that identifier is accessible
- The same identifier may refer to different things in different parts of the program
  - Different scopes for same name don't overlap
- An identifier may have restricted scope

Static vs. Dynamic Scope

- Most languages have static scope
  - Scope depends only on the program text, not run-time behavior
  - Cool has static scope
- A few languages are dynamically scoped
  - Lisp, SNOBOL
  - Lisp has changed to mostly static scoping
  - Scope depends on execution of the program
Static Scoping Example

let x: Int <- 0 in
{ x;
    let x: Int <- 1 in
    x;
    x;
}

Static Scoping Example (Cont.)

let x: Int <- 0 in
{ x;
    let x: Int <- 1 in
    x;
    x;
}

Uses of x refer to closest enclosing definition.

Dynamic Scope

• A dynamically-scoped variable refers to the closest enclosing binding in the execution of the program.

• Example
  g(y) = let a ← 4 in f(3);
  f(x) = a;

• More about dynamic scope later in the course.
Scope in Cool

- Cool identifier bindings are introduced by
  - Class declarations (introduce class names)
  - Method definitions (introduce method names)
  - Let expressions (introduce object id's)
  - Formal parameters (introduce object id's)
  - Attribute definitions in a class (introduce object id's)
  - Case expressions (introduce object id's)

Scope in Cool (Cont.)

- Not all kinds of identifiers follow the most-closely nested rule
- For example, class definitions in Cool
  - Cannot be nested
  - Are globally visible throughout the program
- In other words, a class name can be used before it is defined

Example: Use Before Definition

```plaintext
Class Foo {
  . . . let y: Bar in . . .
};

Class Bar {
  . . .
};
```
More Scope in Cool

Attribute names are local to the class

Class Foo {
    f(): Int { a; }
    a: Int ← 0;
}

More Scope (Cont.)

- Method and attribute names have complex rules
- A method need not be defined in the class in which it is used, but in some parent class
- Methods may also be redefined (overridden)

Implementing the Most-Closely Nested Rule

- Much of semantic analysis can be expressed as a recursive descent of an AST
  - Process an AST node n
  - Process the children of n
  - Finish processing the AST node n
- When performing semantic analysis on a portion of the AST, we need to know which identifiers are defined
Implementing . . . (Cont.)

- Example: the scope of let bindings is one subtree
  
  ```
  let x: Int ← 0 in e
  ```

- `x` is defined in subtree `e`

Symbol Tables

- Consider again: `let x: int ← 0 in e`
- Idea:
  - Before processing `e`, add definition of `x` to current definitions, overriding any other definition of `x`.
  - After processing `e`, remove definition of `x` and restore old definition of `x`.

- A symbol table is a data structure that tracks the current bindings of identifiers.

A Simple Symbol Table Implementation

- Structure is a stack
- Operations
  - `add_symbol(x)` push `x` and associated info, such as `x`'s type, on the stack.
  - `find_symbol(x)` search stack, starting from top, for `x`. Return first `x` found or NULL if none found.
  - `remove_symbol()` pop the stack

- Does this work?
Limitations

- The simple symbol table works for let
  - Symbols added one at a time
  - Declarations are perfectly nested
- Doesn’t work for
  foo(x: Int, x: String):
- Other problems?

A More Powerful Symbol Table

- enter_scope()     start a new nested scope
- find_symbol(x)   finds current x (or null)
- add_symbol(x)    add a symbol x to the table
- check_scope(x)   true if x defined in current scope
- exit_scope()     exit current scope
- A symbol table manager for your project

Class Definitions

- Class names can be used before being defined
- We can’t check that class names
  - using a symbol table
  - or even in one pass
- Solution
  - Pass 1: Gather all class names
  - Pass 2: Do the checking
- Semantic analysis requires multiple passes
  - Probably more than two
Operations on symbol table

- lookup - find certain entry
- insert - add new entry
- delete - remove entry
- enter and exit a scope

Example of Information - static scoped

1. String of characters denoting the name of id
   - sometimes unique number is given to string
   - if same id can be used in more than one declaration
     then indication of which block the variable was declared is needed - nesting level
2. Information for compiler to determine the storage layout and size based upon the type of an element
   - storage for elementary data type: character, integer
   - storage for aggregate types: arrays, structs
   Layout determined when declarations processed
   - relative address or offset is computed with respect to the beginning of storage allocated for the module

3. Attributes of symbols
   kind - label, type, variable, parameter, ...
   data type- array, record, real parameter, subrange
   type of symbol determined after syntactical construct recognized
   Entry in table - (string, unique-id), kind of symbol and attributes

Attributes vary with the kind of symbol
- label - location
- variable - type, address (offset, nesting level)
- type - descriptor of type
Attributes also vary with language - consider arrays
Fortran - fixed number of dimensions Dimension A(100), IF(200)
Arrays can be more complicated

- Type of elements can be records, etc.
- Subscript type is also more complicated - integer sub-range, enumerated type ...

<table>
<thead>
<tr>
<th>id</th>
<th>array</th>
<th>type of element</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td></td>
<td></td>
<td>id</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>lb1</th>
<th>ub1</th>
<th>lb2</th>
<th>ub2</th>
</tr>
</thead>
</table>

type of subscript 1 type of subscript 2

More complexity...

- bounds are not known at compile time - for example, the bounds may be an expression which will be computed at run time
- store pointer to where the values of bounds can be found at run time
- code generated to compute value of bound and store in a temporary & the temporary’s address passed in code to find the upper and lower bounds

Record/structure

<table>
<thead>
<tr>
<th>1st field</th>
<th>2nd field</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>record</td>
</tr>
<tr>
<td>total size</td>
<td>start address</td>
</tr>
<tr>
<td>name</td>
<td>type</td>
</tr>
<tr>
<td>name</td>
<td>type</td>
</tr>
</tbody>
</table>

From start address can compute start addresses of each of the components/fields
How Storage Addresses computed

How are storage addresses computed?
- addresses are offsets since physical addresses not known
- allocate storage for a program module - start at 0
  1st variable - offset 0
  2nd variable - offset 0 + length of variable 1
  etc.

Compute addresses

Loader converts logical addresses to physical addresses
- Each procedure is compiled with an offset
- Each variable is compiled into (nesting level, offset) -
  - nesting level used to find the right storage for module
  - offset is used to find right address of element in module

Symbol table structure

linear list
binary tree
hash table
space  time
trade in complexity: time and space
First ignore scope - assume single program module
Linear List

Self-organizing list - use a linked & move the most recently used entry to beginning
Result: most often access information propagates to the beginning of the list

Binary tree

If balanced search proceeds faster although the tree uses more space than an array/list
In worst case, tree may not be balanced and reduce to a linear list

Hash Table

hash function h(id) -> index into the array
- entries are added when declarations are processed
- look up performed when statements are processed
Adding Scope information

- Separate table for each procedure (i.e., for each scope)
- At a given point have symbol table for each nesting level ≤ current nesting level
- Info is deleted as well as created

Search from bottom
When table not needed, delete the pointer to its symbol table else keep pointers to table - code generation, debugging - delete only from active

Binary tree
Search in current symbol table for id:
- if not found find the previous table and continue in this manner until the id is found
Hash table technique - Single hash table
- Link together different entries for the same identifier and associate nesting level with each occurrence of the same name.
- The first one is the latest occurrence of the name, i.e., highest nesting level

\[
\begin{array}{c|c|c}
\text{id} & \text{level} \\
\hline
h(0) & i & 1 \\
h(k) & k & 2 \\
& k & 1 \\
\end{array}
\]

- nesting level 0
- nesting level 1

During exit from a procedure - delete all entries of the nesting level we are exiting - must be able to do this
Remove links to most recent scope

Three ways to do this
1. Search for the correct items to remove - rehash expensive
2. Use extra pointer in each element to link all items of the same scope
3. Use a stack that has entries for each scope

\[
\begin{array}{c|c|c|c}
\text{level} \\
\hline
1 & 2 & 3 \\
\end{array}
\]

Pop entry and delete from hash table - keep stack entries around