Lecture 6: Symbols and Types

I. Symbol Processing Objectives
   A. Enter all identifiers defined in the program into a symbol table
      1. With appropriate properties
      2. In a manner that they can be used
   B. Associate all references (identifiers) with the appropriate definitions
      1. Identify undefined references
      2. Identify ambiguous references

II. Basic Components
   A. Names
      1. A name is a way of associating a lexical token with a symbol
         • Can be a String
         • This is not optimal (why?)
         • Can use Atoms (String.intern)
      2. Will vary from language-to-language
         • Case sensitive or insensitive
         • Length sensitive or insensitive
   B. Scopes
      1. These are regions that contain symbols
         • Can be logical or physical
      2. Properties
         • Type of scope
            i. TOP, GLOBAL, FILE, CLASS, ENUM, FUNCTION, FUNCTION_BODY, EXCEPTION, LOCAL, NAMESPACE, PACKAGE
            ii. Different types might have different rules
               ➢ For lookup (possibly based on symbol type)
                  ▪ Import definitions do what?
                  ▪ Lookup of name after ‘struct’ in C
               ➢ For definition
                  ▪ Enum definitions
• Parameter definitions
• Other examples?

• Parent Scope
  i Lookup occurs in a given scope (current or explicit)
  ii What happens if the name is not defined there
  iii Lookup continues in the ‘parent’ scope

3. Scoping
• Some languages are simple (C for example) – simple hierarchy
  i Structures yield a scope for looking up fields (not originally)
• Object oriented scoping can be more complex
  i Scopes introduced by classes
  ii With subclasses and inner classes, this can get messy
  iii Consider Java with a subclass defined as an inner class
• What is the parent scope at a point
  i Super class scopes (interfaces as well)
  ii Parent scope for class definition on up
     ➢ This might be another class with super classes
  iii Imported scope
• In C++ also need to consider global scope, imported scopes
  i Different file scopes for the same class
  ii Multiple superclasses (same can appear more than once)
  iii Imported (using) names and ambiguity

4. You want to encapsulate this from the rest of the compiler

C. Symbols
1. Actual objects in the symbol table
   • Represent definitions
   • Contain information about the definition
2. Symbol kind
   • Class, interface, function, method, static function, static method,
     forward function, forward method, forward static function, forward
     static method, variable, argument, field, static field, union field,
     constant, enumerant, type, struct tag, union tag, enum tag, class tag,
     macro, label, package, namespace, reference
   • Determines other properties and behaviors
   • Can have other properties as well (final, volatile,..)
3. Storage type
• Global, file, external reference, routine, local, local static, local external, record, record static, record reference, union, package, parameter, constant
• Determines allocation and access behavior

D. Basic Operations
1. Scope.define(symbol)
   • Defines the symbol in the given scope
   • Creates a name->symbol mapping within scope
   • Complexities
     i. Might allow multiple symbols for the same name
        ➢ Overloaded operations
        ➢ Different name tables (fields vs methods in decaf, struct vs other in C)
        ➢ Might need to define in the parent scope
           ▪ Enumeration constants
           ▪ external local variables
           ▪ anonymous unions
        ➢ Need to check for illegal definitions
           ▪ Duplicate names
           ▪ Duplicate overloadings
           ▪ Illegal redefinitions in subclasses
        ➢ Explicit handling of imported scopes
           ▪ C++ inheritance names
           ▪ Might require removing symbols from the scope
   
2. Scope.lookup(String)
   • Looks up the given symbol in the given scope
   • Use the name->symbol map in the scope to do so
   • Complexities
     i. Need to handle errors
     ii. Might need to return a set of symbols (methods)
        ➢ Or do the overloading here by passing in more information than just the name
        ➢ Or do resolution here based on expected kind, passing information about the expected kind (e.g. method vs field)
     iii. Might want to restrict which parents are looked at
        ➢ Explicit scope reference – no parents
        ➢ Class-based reference – only look at superclasses
III. Type Processing
A. Understanding the type system of the language
   1. Enumerating all the possible types
   2. Representing these types
   3. Manipulating these types
B. Range of types
   1. Primitive types
      • What are the primitive types of java
      • Need to include ‘void’, ‘null’, <type>
      • Need to include <int const> types, <init> types
      • What are enum’s in C/C++
   2. Composed types
      • Array of, pointer to, reference of, pointer to method
      • Pascal arrays
      • Named types
      • Records: struct, union, enum
      • Classes (treat as records?)
      • Method types: function types
         i With or without return type
C. Type Manipulation
   1. Check type compatibility
      • Under different circumstances
         i Assignment, parameters passing, operations, casts, initializers, ...
         ii Result is generally a hierarchy of compatibilities
- C++: exact match, promotion, standard conversions, user-defined conversion, ellipses
  - Might need to understand conversion methods (C++)
    - Check for best match with overloaded argument types

2. Merge two types to get common type
   - `(x ? y : z)` construct

3. Interaction with scoping
   - Find associated scope
   - Lookup field, method

IV. Homework
A. Read 5.1-5.2