Semantic Analysis

The compilation process is driven by the syntactic structure of the program as discovered by the parser

Semantic routines perform *static analysis*:

- interpret meaning of the program based on its syntactic structure
- associated with individual productions of a context free grammar or subtrees of a syntax tree
- two purposes:
  - finish analysis by deriving context-sensitive information
  - begin synthesis by generating the IR or target code

Context-sensitive analysis

What context-sensitive questions might the compiler ask?

1. Is `x` scalar, an array, or a function?
2. Is `x` declared before it is used?
3. Are any names declared but not used?
4. Which declaration of `x` does this reference?
5. Is an expression *type-consistent*?
6. Does the dimension of a reference match the declaration?
7. Where can `x` be stored? (heap, stack, ...)
8. Does `*p` reference the result of a malloc()?
9. Is `x` defined before it is used?
10. Is an array reference *in bounds*?
11. Does function `foo` produce a constant value?
12. Can `p` be implemented as a *memo-function*?

These cannot be answered with a context-free grammar

Context-sensitive analysis

Why is context-sensitive analysis hard?

- answers depend on values, not syntax
- questions and answers involve non-local information
- answers may involve computation

Several alternatives:

abstract syntax tree (*attribute grammars*)	specify non-local computations automatic evaluators
symbol tables	central store for facts express checking code
language design	simplify language avoid problems

Symbol tables

For *compile-time* efficiency, compilers use a *symbol table*:

> associates lexical *names* (symbols) with their *attributes*

What items should be entered?

- variable names
- defined constants
- procedure and function names
- literal constants and strings
- source text labels
- compiler-generated temporaries (*we'll get there*)

Separate table for structure layouts (types) (*field offsets and lengths*)

A symbol table is a compile-time structure

Symbol table information

What kind of information might the compiler need?

- textual name
- data type
- dimension information (*for aggregates*)
- declaring procedure
- lexical level of declaration
- storage class (*base address*)
- offset in storage
- if record, pointer to structure table
- if parameter, by-reference or by-value?
- can it be aliased? to what other names?
- number and type of arguments to functions

Scope

The *scope* of a definition of identifier x is the part of the program where a (non-definition) occurrence of x may refer to this definition.

$\Rightarrow$ semantic analysis must map each occurrence of an identifier to its definition.

Example: Scopes in Java

- public class: entire program
- class: classes in package
- public, (default), protected, private fields
- local variables: just in the enclosing block

Visibility

A definition of an identifier x may be in scope, but not *visible*.

A definition of x is *shadowed* at some program point if there is another intervening enclosing definition of x.

Visibility example

```
class Outer {
  int a, b;               // (1)
  static class P {
    int a, c;             // (2)
                          // def of a at (1) shadowed
                          // def of c at (3) shadowed
  }
  int c, d;               // (3)
  static class Q {
    int a, d;             // (4) shadows (1)a, (3)d
    static class R {
      int a, c;           // (5) shadows (4)a, (3)c
    }
  }
}
```

Nested scopes: block-structured symbol tables

What information is needed?

- when asking about a name, want *most recent* declaration
- declaration may be from current scope or outer scope
- innermost scope overrides outer scope declarations

Nested scopes

Key point: new declarations (usually) occur only in current scope

What operations do we need?

operation	comment	frequency
`void put (Symbol key, Object value)`	bind key to value	rare
`Object get(Symbol key)`	return value bound to key	frequent
`void beginScope()`	remember current state of table	very rare
`void endScope()`	close current scope and restore table to state at most recent open begin-Scope	very rare

Data structure for block-structured symbol table

Idea:

- Each identifier points to a stack of entries pointing to the definitions in scope with the currently visible one at the head.
- These entries have a secondary list structure that connects all entries defined in the same scope.
- A stack of open scopes consisting of entries that contain the entry points of the secondary list structure.

Operations

- `beginScope()` push a new entry on the stack of open scopes
- `put (key, value)` push a new entry on the stack for `key`, insert entry into list of current scope
- `get (key)` obtain top entry from stack for `key`
- `endScope()` pop entry from stack of open scopes, following the list in this entry pop the top entry in each concerned stack

[Intentionally left blank]

Attribute information

Attributes are internal representation of declarations

Symbol table associates names with attributes

Names may have different attributes depending on their meaning:

- variables: type, procedure level, frame offset
- types: type descriptor, data size/alignment
- constants: type, value
- procedures: formals (names/types), result type, block information (local decls.), frame size

Type expressions

Type expressions are a textual representation for types:

1. basic types: *boolean*, *char*, *int*, *float*, etc.
2. type names
3. constructed types (constructors applied to type expressions):
   (a) $array(T)$ denotes array of elements type T
   (potentially, there is also an index type I, e.g., $array(1\ldots10, integer)$)
   (b) classes: fields have names and visibilities
   e.g., $class((\texttt{a} : int), (\texttt{b} : float))$
   (c) $D \rightarrow R$ denotes type of method mapping domain D to range R
   e.g., $int \times int \rightarrow int$

Type compatibility

Type checking needs to determine type equivalence

Two approaches:

Name equivalence: each type name is a distinct type

Structural equivalence: two types are equivalent iff they have the same structure (after substituting type expressions for type names)

- $s \equiv t$ iff. s and t are the same basic types
- $array(s) \equiv array(t)$ iff. $s \equiv t$
- $s_1 \times s_2 \equiv t_1 \times t_2$ iff. $s_1 \equiv t_1$ and $s_2 \equiv t_2$
- $s_1 \rightarrow s_2 \equiv t_1 \rightarrow t_2$ iff. $s_1 \equiv t_1$ and $s_2 \equiv t_2$

Java inheritance: field shadowing

- Fields declared in a subclass can *shadow* fields declared in superclasses
- Consider:

```
class A { int j; }
class B extends A { int j; }

A a = new A();// let's call this object X
              // X has one field, named j, declared in A
a.j = 1;      // assigns 1 to the field j of X declared in A
a = new B();  // let's call this object Y
              // Y has two fields, both named j,
              // one declared in A, the other in B
a.j = 2;      // assigns 2 to the field j of Y declared in A
B b = a;
b.j = 3;      // assigns 3 to the field j of Y declared in B
```

Java inheritance: method overriding

- Methods declared in subclasses can *override* methods declared in superclasses
- Overriding is same name used to name a different thing, regardless of context, such as methods in subclasses with the same name
- Consider:

```
class A { int j; void set_j(int i) { this.j = i; }
class B extends A { int j; void set_j(int i) { this.j = i; }

A a = new A();// let's call this object X
a.set_j(1);   // assigns 1 to the field j of X declared in A
              // i.e., invokes A set_j method
a = new B();  // let's call this object Y
a.set_j(2);   // assigns 2 to the field j of Y declared in B
              // i.e., invokes B set_j method
B b = a;
b.set_j(3);   // assigns 3 to the field j of Y declared in B
              // i.e. invokes B set_j method
```

Java method overloading

- Java also supports method overloading, which has nothing to do with inheritance

- In Java, the “name” of a method includes the number and the types of the method’s arguments.

- Consider:

```
class A {
  int j;
  boolean b;
  void set(int i) { this.j = i; }
  void set(boolean b) { this.j = b; }
}
```

- Don’t confuse method overloading with method overriding