Semantic Analysis

*The context-free structure discovered by the parser is **not** sufficient for compilation*

Semantic routines

- perform *static analysis* by analyzing the meaning of the program without executing it
- are 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

1. What is the type of `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. Where can `x` be stored? (its *storage classe*: heap, stack, ...)
7. Does `*p` reference the result of a malloc()?
8. Is `x` defined before it is used?
9. Is an array reference *in bounds*?
10. Does an expression produce a constant value?

These questions 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

Major ingredients of most analyses

walk of abstract syntax tree	specify non-local computations
(*attribute grammars*)	automatic evaluators
symbol tables	central store for facts
	explicit checking code

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*)

Symbol table information — Attributes

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 (*for functions*)

Attribute information

Attributes: 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

Scope

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

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

Some languages enable access to shadowed definitions

- using qualification $C.D.x$
- using imports and scope management (perhaps even renaming and hiding definitions)

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

Which information is needed?

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

Nested scopes

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

Which 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 beginScope	very rare

Naive implementations

- List / stack of `beginScope` and `put`: `get` is $O(n)$
- List / stack of search trees or hash tables: `get` is $O(\log n)$

Goal: `get` should be $O(1)$

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]

Example for attributes: Type expressions

Type expressions are a textual representation for types:

1. basic types: $boolean$, $char$, int, $float$, etc.
2. 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.,
   $array(1 \ldots 10, integer)$)
   (b) classes and interfaces: fields and methods have names,
   visibilities, static vs dynamic, types
   e.g., $class(\texttt{a} : int, \texttt{b} : float)$
   (c) $D \to R$ denotes type of method mapping domain D to range R
   e.g., $int \times int \to int$
3. type variables (e.g., Java generics)
4. type constraints (e.g., `implements Comparable<A>`)

Type compatibility

Type checking needs to determine *assignment compatibility*

Figure out the static type of the arguments of an operator or method.

The static type must be assignment compatible to the expected type.

Rules for type compatibility in Java

1. Every type is assignment-compatible with itself.
2. The boolean type is not assignment-compatible with any other type.
3. A value of any integer type can be assigned to a variable of any other integer type if the variable is of a type that allows it to contain the value without any loss of information.
4. A value of any integer type can be assigned to a variable of any floating-point type, but a value of any floating-point type cannot be assigned to a variable of any integer type.
5. A float value can be assigned to a double variable, but a double value cannot be assigned to a float variable.
6. With a type cast, a value of any arithmetic type can be assigned to a variable of any other arithmetic type.
7. Any reference can be assigned to a variable that is declared of type Object.

8. A reference to an object can be assigned to a class-type reference variable if the class of the variable is the same class or a superclass of the class of the object.
9. A reference to an object can be assigned to an interface-type reference variable if the class of the object implements the interface.
10. A reference to an array can be assigned to an array variable if either of the following conditions is true:
    - Both array types contain elements of the same type.
    - Both array types contain object references and the type of reference contained in the elements of the array reference can be assigned to the type of reference contained in the elements of the variable.

Java inheritance: field shadowing

- Fields declared in a subclass can *shadow* fields declared in superclasses
- Shadowing is redefining a name in a subsidiary scope
- Field shadowing is *resolved at compile time* according to the *static type* of the object
- Shadowed fields are still accessible
- (some languages disallow field shadowing)

Java inheritance: field shadowing example

```
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
B b = new B();// let's call this object Y
              // Y has two fields, both named j,
              // one declared in A, the other in B
a = b;        // change static type to A
a.j = 2;      // assigns 2 to the field j of Y declared in 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 *resolved at run time* according to the *run-time type* of the object
- Overridden methods are only accessible from the overriding method through `super` calls

Java inheritance: method overriding example

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

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

b.set_j(3);   // assigns 3 to the field jb of Y declared in B
              // i.e. invokes B.set_j method
```

Java method overloading

- Java also supports method overloading, which binds the same name to multiple “things” in the same scope
- Overloading is *resolved at compile time* according to the *context*: for a method, the context consists of the (static) argument types (and the result type)
- Overloading has *nothing* to do with inheritance
- Don’t confuse method overloading with method overriding
- Arithmetic operators are often overloaded
- Method `set` is overloaded in this example:

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

Resolution of overloading

Complicated in Java because of interaction with subtyping

```
class A {}
class B extends A {}
class C {}
class D extends C {}

static void m (B b, C c) {...}   // (1)
static void m (A a, D d) {...}   // (2)

// Java chooses the best matching declaration for the
// static argument types
m(new B(), new D());     // ??? matches (1) and (2) -> error

// adding this declaration resolves the error
static void m (B b, D d) {...}
```

Resolution of overloading *without* subtyping

Consider expression trees generated by $E \rightarrow Op\,E^*$, where Op encompasses operators, functions, and constants.

For each node of the tree with operator op collect the following information

- $nrChildren$, a property of the expression tree
- $child(i)$, access the ith child
- vis, the set of visible definitions of op; computed before overloading resolution
- ops_in, set of candidate definitions
- ops_bu, first improvement of that set
- ops_td, second improvement of that set
- $valid$, overloading resolved: expression is unambiguous

Resolution of overloading, continued

For each definition d of an overloaded operator

- $rank(d)$, the arity of this operator definition
- $res\_type(d)$, the result type
- $par\_type(d,i)$, the type of the ith parameter

That is, if $m = rank(d)$, then

$$op(d) : par\_type(d,1) \times \ldots \times par\_type(d,m) \rightarrow res\_type(d)$$

Resolution of overloading, algorithm

1. For each node n, compute

$$ops\_in(n) = \{d \in vis(n) \mid rank(d) = nrChildren(n)\}$$

2. During a bottom-up traversal, compute for each node n

$$ops\_bu(n) = \{d \in ops\_in(n) \mid \begin{array}{l} \forall 1 \leq i \leq rank(d). \\ \exists d_i \in ops\_bu(child(n,i)). \\ par\_type(d,i) = res\_type(d_i)\} \end{array}$$

3. During a top-down traversal, compute at each node n, for each child i

$$ops\_td(child(n,i)) = \{d_i \in ops\_bu(child(n,i)) \mid \begin{array}{l} \forall d \in ops\_td(n). \\ res\_type(d_i) = par\_type(d,i)\} \end{array}$$

4. During a bottom-up traversal, compute at each node n

$$valid(n) = \|ops\_td\| = 1$$

Resolution of overloading, example

Consider the following definitions

$$
\begin{array}{ll}
f: & A \times C \to C \\
f: & A \times B \to C \\
f: & B \times A \to D \\
g: & A \times C \to C \\
g: & C \to C \\
g: & D \to D \\
x: & A \\
x: & B \\
h: & C \to R
\end{array}
$$

and apply the algorithm to resolve overloading in

$$h(g(f(x,x)))$$