Semantic Analysis

Is Syntax Analysis Enough?

Parsing cannot catch some errors:

- Some language constructs are not context-free
- Example: identifiers are declared before use

Add a **semantic analysis** phase to find remaining problem as the last phase of the front end.

Semantic analyzer checks:

- All identifiers are declared before use
- Type consistence
- Inheritance relationship is correct
- A class is defined only once
- A method in a class is defined only once
- Reserved identifiers are not misused
- …

Matching Declarations with Uses

We must do this in most languages (static languages):

```
void foo() {
        char x;
        ...
        {
                int x;
                ...
        }
        x = x + 1;
}
```

Which x do we match in the `x = x + 1;` statement?

Scope

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

Two types: static scope and dynamic scope

Static Scope

Static scope depends on the program text, not run-time behavior

- Most languages have static scope
- Refer to the closest enclosing definition

```
void foo() {
        char x;
        ...
        {
                int x;
                ...
        }
        x = x + 1;
}
```

Dynamic Scope

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

```
#!/usr/bin/perl
use strict;
use warnings;

&foo;

sub foo {
        my $a = 3;
        &bar;
}

sub bar {
        print $a;
}
```

```
#!/usr/bin/perl
use strict;
use warnings;

&foo;

sub foo {
        local $a = 3;
        &bar;
}

sub bar {
        print $a;
}
```

Dynamic Scope

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

```
#!/usr/bin/perl
use strict;
use warnings;

&foo;

sub foo {
        my $a = 3;
        &bar;
}

sub bar {
        print $a;
}
```

```
Name "main::a" used only once: possible typo at ex.pl line 13.
Use of uninitialized value $a in print at ex.pl line 13.
```

Dynamic Scope

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

```
#!/usr/bin/perl
use strict;
use warnings;

&foo;

sub foo {
        local $a = 3;
        &bar;
}

sub bar {
        print $a;
}
```

```
3
```

Tracking Static Scope

We finally need to construct our other major data structure: the **symbol table**.

A symbol table holds a mapping between identifiers (symbols) and their

- Types (size and interpretation)
- Locations (declarations/uses and line numbers)

A **use** is a non-defining occurrence of the identifier.

Symbol tables reflect **environments**, which are sets of bindings from an identifier to its meaning.

We'll use the notation $\sigma_0 = \{x \mapsto \text{int}, s \mapsto \text{String}\}$ to indicate that there is some environment, σ_0, with the identifiers x and s with types int and String respecitively.

Example

```
public class Example {
    int b;

    public int f(int a) {
        int b = 4;
        int c = a + b;
        return c;
    }
}
```

There exists some initial environment, σ_0, that contains information about the things that enclose class `Example`.

For instance, we implicitly extend `Object` and that must be defined in σ_0.

`Example` then defines a new environment, that is the combination of σ_0 and `Example`:

$$\sigma_1 = \sigma_0 + \{b \mapsto \text{int}, f \mapsto \{\text{int}, (\text{int})\}\}$$

`f` defines:

$$\sigma_2 = \sigma_1 + \{a \mapsto \text{int}, b \mapsto \text{int}, c \mapsto \text{int}\}$$

But what is the meaning of the + operator for environments?

Combining Environments

When we try to combine our environments from `Example` and `f`, we get a problem:

$$\{b \mapsto \text{int}, f \mapsto \{\text{int}, (\text{int})\}\} + \{a \mapsto \text{int}, b \mapsto \text{int}, c \mapsto \text{int}\}$$

There is a conflict between the identifiers b in both scopes. When we write:

```
int c = a + b;
```

which b do we want?

We have already indicated that we want the most recent declaration in the nearest enclosing scope, and that we say that `f`'s b **shadows** the b in `Example`.

That means that + is not commutative, $\sigma_X + \sigma_Y$ is different than $\sigma_Y + \sigma_X$

Implementing Environments

Two basic strategies to keep track of the changes that each scope makes.

Functional Style

- Keep σ_1 unchanged while we create σ_2 and σ_3

Imperative Style

- Destructively modify σ_1 until it becomes σ_2
- While σ_2 exists, we cannot look things up in σ_1
- When we are done with σ_2, undo the modifications to get σ_1 back

Either style of environment management can be used regardless of whether the language being compiled is functional, imperative, or OOP.

Data Structures for Symbol Tables

We have an unknown amount of information that will need to be searched, inserted, and organized.

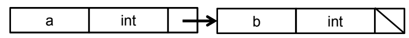
The usual data structure suspects:

- Array
- List
- Tree
- Hash table

Consider each for only a single environment. What operations will we need?

- Create a new symbol
- Lookup a symbol
- Delete the structure

Linked List of Symbols



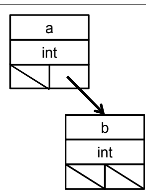
Arrays are not good for insertion and so we may consider instead a linked list.

They have the same search cost (O(n)) but allow for easy insertion and removal.

One possible optimization to do is to move the element you find after a search to the start of the list.

Then subsequent lookups of the most frequent identifiers will be fast.

Binary Search Tree

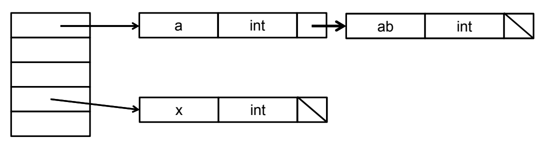


Could build a binary search tree to quickly find identifier names.

Uses a bit more space for the additional pointers.

May be no better than the linked list if the tree is unbalanced.

Hashtable



Use a hash function to index a table whose contents point to a linked list of elements that hashed there (closed addressing or chaining)

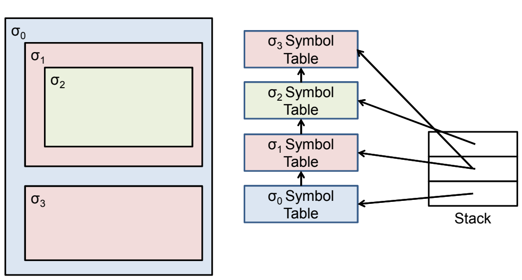
Hash function of input is computable in O(1) time, so search is fast.

Table needs to be much larger than the input to avoid too many collisions making the chain too large.

Multiple Scopes

We don't want to delete information out of the symbol tables, but we still must deal with shadowing.

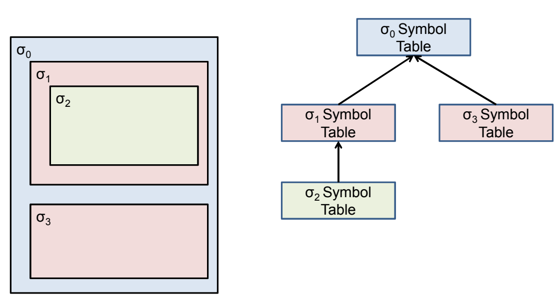
Can use a stack to manage which scopes are active currently



Multiple Scopes

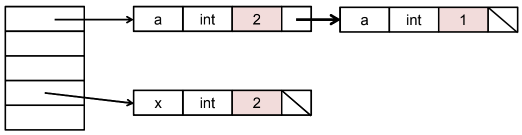
Or we can make a tree of symbol tables.

If we do not find a symbol in one, we can go up to the parent.



Multiple Scopes

Add nesting level to elements in the hashtable



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
- When exiting level k, remove all symbols with level k
- Inconvenient for dot access (Class.Func)

Type Checking

Type checking will proceed in two passes:

1. Build the symbol table (probably from a stack or tree of hashtables)
2. Perform the semantic analysis

Why can't we do both at once?

```
class A {
    B b;
}

class B {
    A a;
}
```

In languages like Java, we can have co-defined types, so we must find the types before we can check them.

Symbol Table Entries

What constitutes a symbol? What information would we need to keep about each symbol?

This is language dependent. In MiniJava:

- Identifiers come from class names, method names, and variable names

- Methods are bound to their signatures (return type and parameter list)
- Local variables are bound to the methods they're declared in
- Variable and formal parameter names are bound to their type.

- Class names should be bound to their member variables and methods

Creating a class means creating a new type.

Phase 1: Build the table

We can construct a `BuildSymbolTableVisitor` which visits each node in the AST.

For class declarations, we add a new entry to the top-level (what we called σ_0) symbol table. (MiniJava does not support inner classes.)

For method declarations, we add entries to the class with the signature of the method.

For parameters and variables, we add them to the appropriate symbol table at the appropriate nesting.

This visitor can detect certain errors, most notably redeclaration

Phase 2: Check the Types

Create a `TypeCheckVisitor` that walks the AST again. Its `visit` method returns a representation of the type of the expression so that we can forward that information to parent nodes in the tree.

Examine each statement and expression:

- If it is a binary operator, check that the left and right hand side are compatible
  - Could be the same type or one might be coerced to the other
  - Could be a subclass relationship
- Method names must exist in the class
- Method actual parameter number and types must be matched
- Method returns a typed-value or void
- Class member variables must exist and yield the proper type

Errors

Report errors and continue on so that more than one message can be displayed per compilation attempt.

That may mean adding invalid symbols to the symbol table just to be able to continue.

The output of the semantic analysis phase should be a valid program in some intermediate representation so that later phases do not need to do as much error checking.