

Overview of Semantic Analysis

Adapted from Lectures by
Prof. Alex Aiken and George Necula
(UCB)

CS780(Prasad)

L19SA

1

The Compiler So Far

- Lexical analysis
 - Detects inputs with illegal tokens
- Parsing
 - Detects inputs with ill-formed parse trees
- Semantic analysis
 - Last "front end" phase
 - Catches all remaining errors

CS780(Prasad)

L19SA

2

Why a Separate Semantic Analysis?

- Parsing cannot catch some errors.
- Some language constructs are not context-free.
- **Static Check:**
 - Identifier declaration and use
 - An abstract version of the problem is:
 $\{ w c w \mid w \in (a + b)^* \}$
 - The 1st w represents a declaration; the 2nd w represents a use.
- **Dynamic Check:**
 - Array bounds check
 - Null pointer dereference check

CS780(Prasad)

L19SA

3

What Does Semantic Analysis Do?

Checks of many kinds . . .

coolc checks:

1. All identifiers are declared.
2. Types compatibility.
3. Inheritance relationships.
4. Classes defined only once.
5. Methods in a class defined only once.
6. Reserved identifiers are not misused.

...

CS780(Prasad)

L19SA

4

More on Semantic Checks

Establish that a program conforms to language definition. (Requirements language dependent)

- **Flow of control checks**
 - Declaration of a variable should be *before* use.
 - Each exit path returns a value of the correct type.
- **Uniqueness Checks**
 - No identifier can be used for two different definitions in the same scope.

CS780(Prasad)

L19SA

5

Type checks

- Number of arguments **matches** the number of formals and the corresponding types are **equivalent**.
- Each access of a variable should **match** the declaration (arrays, structures etc.).
- Identifiers in an expression should be **"evaluatable"**.
- LHS of an assignment should be **"assignable"**.
- In an expression, all the types of variables, method return types and operators should be **"compatible"**.

CS780(Prasad)

L19SA

6

Scope

- Matching identifier declarations with uses.
 - Important static analysis step in most languages.
 - Including COOL!

- **What's Wrong?**

- Example 1

```
Let y: String ← "abc" in y + 3
```

- Example 2

```
Let y: Int in x + 3
```

CS780(Prasad)

L19SA

7

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.
- An identifier may have restricted scope.

CS780(Prasad)

L19SA

8

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.

CS780(Prasad)

L19SA

9

Static Scoping Example

```
let x: Int ← 0 in
{
  x;
  let x: Int ← 1 in
  x;
}
```

Uses of *x* refer to **closest enclosing definition**.

CS780(Prasad)

L19SA

10

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;
```

CS780(Prasad)

L19SA

11

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)

CS780(Prasad)

L19SA

12

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.

CS780(Prasad)

L19SA

13

Example: Use Before Definition

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

CS780(Prasad)

L19SA

14

More Scope in Cool

Attribute names are global within the method in which they are defined.

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

CS780(Prasad)

L19SA

15

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).

CS780(Prasad)

L19SA

16

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.

CS780(Prasad)

L19SA

17

Implementing . . . (Cont.)

- Example: the scope of `let` bindings is one subtree

```
let x: Int ← 0 in e
```

- x is defined in subtree e .

CS780(Prasad)

L19SA

18

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.

CS780(Prasad)

L19SA

19

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
- Why does this work?

CS780(Prasad)

L19SA

20

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?

CS780(Prasad)

L19SA

21

A Fancier 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

CS780(Prasad)

L19SA

22

Class Definitions

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

CS780(Prasad)

L19SA

23

Types

- What is a **type**?
 - The notion varies from language to language.
- Consensus
 - A set of values.
 - A set of operations on those values.
- Classes are one instantiation of the modern notion of type.

CS780(Prasad)

L19SA

24

Why Do We Need Type Systems?

Consider the assembly language fragment

```
addi $r1, $r2, $r3
```

What are the types of `$r1`, `$r2`, `$r3`?

Types and Operations

- Certain operations are legal for values of each type.
 - It doesn't make sense to add a function pointer and an integer in C.
 - It does make sense to add two integers.
 - But both have the same assembly language implementation!

Type Systems

- A language's type system specifies which operations are valid for which types.
- The goal of **type checking** is to ensure that operations are used with the correct types.
 - Enforces intended interpretation of values, because nothing else will!

Type Checking Overview

- Three kinds of languages:
 - **Statically typed**: All or almost all checking of types is done as part of compilation (C, Java, Cool).
 - **Dynamically typed**: Almost all checking of types is done as part of program execution (Scheme).
 - **Untyped**: No type checking (machine code).

The Type Wars

- Competing views on static vs. dynamic typing.
- **Static typing** proponents say:
 - Static checking catches many programming errors at compile time.
 - Avoids overhead of runtime type checks.
- **Dynamic typing** proponents say:
 - Static type systems are restrictive.
 - Rapid prototyping difficult within a static type system.

The Type Wars (Cont.)

- In practice, most code is written in statically typed languages with an "escape" mechanism
 - Unsafe casts in C, Java.
- It's debatable whether this compromise represents the best or worst of both worlds.

A simple typed language

- A language that has a sequence of declarations followed by a single expression

```
P → D; E
D → D; D | id : T
T → char | integer | array [ num ] of T
E → literal | num | id | E + E | E [ E ]
```

- Example Program

```
var: integer;
var + 1023
```

CS780(Prasad)

L19SA

31

A simple typed language

- A language that has a sequence of declarations followed by a single expression

```
P → D; E
D → D; D | id : T
T → char | integer | array [ num ] of T
E → literal | num | id | E + E | E [ E ]
```

- What are the semantic rules of this language?

CS780(Prasad)

L19SA

32

Parser actions

```
P → D; E
D → D; D
D → id : T      { addtype(id.entry, T.type); }
T → char       { T.type = char; }
T → integer    { T.type = integer; }
T → array [ num ] of T1
               { T.type = array(T1.type, num.val); }
```

CS780(Prasad)

L19SA

33

Parser actions

```
E → literal    { E.type = char; }
E → num        { E.type = integer; }
E → id         { E.type = lookup_type(id.name); }
E → E1 + E2  { if E1.type == integer and
                  E2.type == integer then
                    E.type = integer
                  else
                    E.type = type_error
                  }
```

CS780(Prasad)

L19SA

34

Parser actions

```
E → E1[E2]  { if E2.type == integer and
                  E1.type == array(s, t) then
                    E.type = s
                  else
                    E.type = type_error
                  }
```

CS780(Prasad)

L19SA

35