# Overview of Semantic Analysis

Lecture 11

# Outline of today's lecture

- The role of semantic analysis in a compiler
- Scope
  - Static vs. dynamic
  - Implementation: symbol tables
- Types
  - Static vs. dynamic

# Topics covered so far

- Regular expressions and regular languages
- Deterministic and non-deterministic finite automata
- Lexers
- Context-free language and context-free grammars
- Parsers
  - Top-down
  - Bottom-up

# Bottom-up vs. Top-down

| | Top-down Parsers | Bottom-up Parsers |
|---|---|---|
| Successful Parse | From start symbol of grammar to the string | From the string to the start symbol of the grammar |
| Example of grammars | LL(k)<br>Left-to-right, Leftmost derivation first | LR(k), LALR<br>Left-to-right, Rightmost derivation first (in reverse) |
| Example of parser technique | Recursive-descent | Shift-reduce |
| Ease of implementation | Literally recursive descent | Many grammar generators available (Yacc, Bison,…) |
| Issue with left-recursion | Yes | No |
| Issues with left-factoring | Yes | No |

# Why a Separate Semantic Analysis?

- Parsing alone cannot catch many errors
- Some language constructs are not context-free
- Separation of concerns: less-complicated parsers

# What Does Semantic Analysis Do?

- Checks of many kinds . . . Checks:

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

    And others . . .

# What's Wrong?

- Example 1

  Let y: String ← "abc" in y + 3

- Example 2

  Let y: Int in x + 3

*Note: An example property that is not context free.*

# Scope

- Matching identifier declarations with uses
  - Important static analysis step in most languages
  - Very useful in catching lots of common programming errors

# 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
  - Different scopes for same name don't overlap

- An identifier may have restricted scope

# Static vs. Dynamic Scope

- Most languages have *static* scope
  - Scope depends only on the program text, not run-time behavior
  - C programming language 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

# Static Scoping Example

```
let x: Int <- 0 in
  {
      x;
      let x: Int <- 1 in
            x;
      x;
  }
```

# Static Scoping Example (Cont.)

```
let x: Int <- 0 in
  {
      x;
      let x: Int <- 1 in
          x;
      x;
  }
```

Uses of x refer to closest enclosing definition

# 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;
```

- More about dynamic scope later in the course

# Scope in Cool

- Cool identifier bindings are introduced by
  - Class declarations (introduce class names)
  - Method definitions (introduce method names)
  - Let expressions (introduce object ids)
  - Formal parameters (introduce object ids)
  - Attribute definitions (introduce object ids)
  - Case expressions (introduce object ids)

# 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

# Example: Use Before Definition

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

Class Bar {
   . . .
};
```

# More Scope in Cool

Attribute names are global within the class in which they are defined

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

# More Scope (Cont.)

- Method/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)

# Implementing the Most-Closely Nested Rule

- Much of semantic analysis can be expressed as a recursive descent of an AST
  - *Before*: Process an AST node *n*
  - *Recurse*: Process the children of *n*
  - *After*: Finish processing the AST node *n*
- When performing semantic analysis on a portion of the the AST, we need to know which identifiers are defined

# Implementing . . . (Cont.)

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

$$\text{let } x\text{: Int} \leftarrow 0 \text{ in } e$$

- x is defined in subtree e

# Symbol Tables

- Consider again: let x: Int $\leftarrow$ 0 in e
- Idea:
  - *Before* processing e, add definition of x to current definitions, overriding any other definition of x
  - *Recurse*
  - *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

# A Simple Symbol Table Implementation

- Use stack that is remains random access for the find operation

- 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?

# Limitations

- The simple symbol table works for let
  - Symbols added one at a time
  - Declarations are perfectly nested

# 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

# Class Definitions

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

# 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

# Why Do We Need Type Systems?

Consider the assembly language fragment

add $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, code written in statically typed languages usually has an “escape” mechanism
  - Unsafe casts in C, Java

- It’s debatable whether this compromise represents the best or worst of both worlds

# Types Outline

- Type concepts in COOL
- Notation for type rules
  - Logical rules of inference
- COOL type rules
- General properties of type systems

# Cool Types

- The types are:
  - Class Names
  - SELF_TYPE

- The user declares types for identifiers

- The compiler infers types for expressions
  - Infers a type for *every* expression

# Type Checking and Type Inference

- *Type Checking* is the process of verifying fully typed programs

- *Type Inference* is the process of filling in missing type information

- The two are different, but the terms are often used interchangeably

# Rules of Inference

- We have seen two examples of formal notation specifying parts of a compiler
  - Regular expressions
  - Context-free grammars

- The appropriate formalism for type checking is logical rules of inference

# Why Rules of Inference?

- Inference rules have the form

  *If Hypothesis is true, then Conclusion is true*

- Type checking computes via reasoning

  *If $E_1$ and $E_2$ have certain types, then $E_3$ has a certain type*

- Rules of inference are a compact notation for "If-Then" statements

# From English to an Inference Rule

- The notation is easy to read with practice
- Start with a simplified system and gradually add features
- Building blocks
  - Symbol $\wedge$ is "and"
  - Symbol $\Rightarrow$ is "if-then"
  - $x:T$ is "$x$ has type $T$"

# From English to an Inference Rule (2)

If $e_1$ has type Int and $e_2$ has type Int,
then $e_1 + e_2$ has type Int

$(e_1 \text{ has type Int} \wedge e_2 \text{ has type Int}) \Rightarrow$
$e_1 + e_2$ has type Int

$(e_1\text{: Int} \wedge e_2\text{: Int}) \Rightarrow e_1 + e_2\text{: Int}$

# From English to an Inference Rule (3)

The statement

$$(e_1\text{: Int} \wedge e_2\text{: Int}) \Rightarrow e_1 + e_2\text{: Int}$$

is a special case of

$$\text{Hypothesis}_1 \wedge \ldots \wedge \text{Hypothesis}_n \Rightarrow \text{Conclusion}$$

This is an inference rule.

# Notation for Inference Rules

- By tradition inference rules are written

$$\frac{\text{` Hypothesis … ` Hypothesis}}{\text{` Conclusion}}$$

- Cool type rules have hypotheses and conclusions

$$\text{` } e:T$$

- ` means “it is provable that . . .”

# Two Rules

$$\frac{i \text{ is an integer literal}}{\vdash i : \text{Int}} \quad [\text{Int}]$$

$$\frac{\vdash e_1 : \text{Int} \qquad \vdash e_2 : \text{Int}}{\vdash e_1 + e_2 : \text{Int}} \quad [\text{Add}]$$

# Two Rules (Cont.)

- These rules give templates describing how to type integers and + expressions

- By filling in the templates, we can produce complete typings for expressions

# Example: 1 + 2

$$\dfrac{\dfrac{1 \text{ is an int literal}}{\text{`}\ 1 : \text{Int}} \qquad \dfrac{2 \text{ is an int literal}}{\text{`}\ 2 : \text{Int}}}{\text{`}\ 1 + 2 : \text{Int}}$$

# Soundness

- A type system is *sound* if
  - Whenever ` e: T
  - Then e evaluates to a value of type T

- We only want sound rules
  - But some sound rules are better than others:

i is an integer literal
—————————————
` i : Object

# Type Checking Proofs

- Type checking proves facts e: T
  - Proof is on the structure of the AST (Abstract syntax tree)
  - Proof has the shape of the AST
  - One type rule is used for each AST node
- In the type rule used for a node e:
  - Hypotheses are the proofs of types of e's sub-expressions
  - Conclusion is the type of e
- Types are computed in a bottom-up pass over the AST

# Rules for Constants

$$\frac{}{\text{` false : Bool}} \quad \text{[False]}$$

$$\frac{\text{s is a string literal}}{\text{` s: String}} \quad \text{[String]}$$

# Rule for New

new T produces an object of type T

- Ignore SELF_TYPE for now . . .

$$\frac{}{\vdash \text{new } T : T} \quad [\text{New}]$$

# Two More Rules

` e:  Bool

---

` !e : Bool    [Not]

` $e_1$: Bool

` $e_2$:T

---

` while $e_1$ loop $e_2$ pool:Object    [Loop]

# A Problem

- What is the type of a variable reference?

$$\frac{x \text{ is a variable}}{\vdash x: ?} \quad [\text{Var}]$$

- The local, structural rule does not carry enough information to give x a type.

# A Solution

- Put more information in the rules!
- A *type environment* gives types for *free* variables
  - A type environment is a function from ObjectIdentifiers to Types
  - A variable is free in an expression if it is not defined within the expression

# Type Environments

Let O be a function from ObjectIdentifiers to Types

The sentence

O ` e: T

is read: Under the assumption that variables have the types given by O, it is provable that the expression e has the type T

# Modified Rules

The type environment is added to the earlier rules:

$$\frac{i \text{ is an integer literal}}{O \vdash i : \text{Int}} \quad [\text{Int}]$$

$$\frac{O \vdash e_1 : \text{Int} \qquad O \vdash e_2 : \text{Int}}{O \vdash e_1 + e_2 : \text{Int}} \quad [\text{Add}]$$

# New Rules

And we can write new rules:

$$\frac{O(x) = T}{\vdash x : T} \quad [\text{Var}]$$

# Let

$$\frac{O[T_0/x] \vdash e_1 : T_1}{O \vdash \text{let } x{:}T_0 \text{ in } e_1 : T_1} \quad \text{[Let-No-Init]}$$

$O[T/y]$ means $O$ modified to return $T$ on argument $y$

Note that the let-rule enforces variable scope

# Notes

- The type environment gives types to the free identifiers in the current scope

- The type environment is passed down the AST from the root towards the leaves

- Types are computed up the AST from the leaves towards the root

# Let with Initialization

Now consider let with initialization:

$$\frac{\begin{array}{c} O \text{ ` } e_0 : T_0 \\ O[T_0/x] \text{ ` } e_1 : T_1 \end{array}}{O \text{ ` } \text{let } x : T_0 \leftarrow e_0 \text{ in } e_1 : T} \quad \text{[Let-Init]}$$

This rule is weak. Why?

# Subtyping

- Define a relation $\leq$ on classes
  - $X \leq X$
  - $X \leq Y$ if $X$ inherits from $Y$
  - $X \leq Z$ if $X \leq Y$ and $Y \leq Z$

- An improvement

$$\frac{\begin{array}{c} O \vdash e_0 : T_0 \\ O[T/x] \vdash e_1 : T_1 \\ T_0 \leq T \end{array}}{O \vdash \text{let } x : T \leftarrow e_0 \text{ in } e_1 : T_1} \quad \text{[Let-Init]}$$

# Assignment

- Both let rules are sound, but more programs typecheck with the second one

- More uses of subtyping:

$$\frac{O(x) = T_0 \quad O \vdash e_1 : T_1 \quad T_1 \leq T_0}{O \vdash x \leftarrow e_1 : T_1} \quad \text{[Assign]}$$

# Initialized Attributes

- Let $O_C(x) = T$ for all attributes $x{:}T$ in class $C$

- Attribute initialization is similar to let, except for the scope of names

$$\frac{\begin{array}{c} O_C(x) = T_0 \\ O_C \vdash e_1 : T_1 \\ T_1 \leq T_0 \end{array}}{O_C \vdash x{:}T_0 \leftarrow e_1;} \quad \text{[Attr-Init]}$$

# If-Then-Else

- Consider:
  if $e_0$ then $e_1$ else $e_2$ fi

- The result can be either $e_1$ or $e_2$

- The type is either $e_1$'s type of $e_2$'s type

- The best we can do is the smallest supertype larger than the type of $e_1$ or $e_2$

# Least Upper Bounds

- $lub(X,Y)$, the least upper bound of $X$ and $Y$, is $Z$ if
  - $X \leq Z \wedge Y \leq Z$

    $Z$ is an upper bound
  - $X \leq Z' \wedge Y \leq Z' \Rightarrow Z \leq Z'$

    $Z$ is least among upper bounds
- In COOL, the least upper bound of two types is their least common ancestor in the inheritance tree

# If-Then-Else Revisited

$$\frac{\begin{array}{c} O \text{ ` } e_0 : \text{Bool} \\ O \text{ ` } e_1 : T_1 \\ O \text{ ` } e_2 : T_2 \end{array}}{O \text{ ` } \text{if } e_0 \text{ then } e_1 \text{ else } e_2 \text{ fi} : \text{lub}(T_1, T_2)}$$

[If-Then-Else]

# Case

- The rule for case expressions takes a lub over all branches

$$
\dfrac{\begin{array}{c} O \text{ ` } e_0 : T_0 \\ O[T_1/x_1] \text{ ` } e_1 : T_{1'} \\ \ldots \\ O[T_n/x_n] \text{ ` } e_n : T_{n'} \end{array}}{O \text{ ` } \text{case } e_0 \text{ of } x_1{:}T_1 \to e_1; \ldots; x_n{:}T_n \to e_n; \text{ esac} : \text{lub}(T_{1'}, \ldots, T_{n'})} \quad [\text{Case}]
$$

# Method Dispatch

- There is a problem with type checking method calls:

$$\frac{\begin{array}{c} O \text{ ` } e_0 : T_0 \\ O \text{ ` } e_1 : T_1 \\ \ldots \\ O \text{ ` } e_n : T_n \end{array}}{O \text{ ` } e_0.f(e_1, \ldots, e_n) : ?} \quad \text{[Dispatch]}$$

# Notes on Dispatch

- In Cool, method and object identifiers live in different name spaces
  - A method foo and an object foo can coexist in the same scope
- In the type rules, this is reflected by a separate mapping $M$ for method signatures

$$M(C,f) = (T_1, \ldots T_n, T_{n+1})$$

means in class $C$ there is a method $f$

$$f(x_1:T_1, \ldots, x_n:T_n): T_{n+1}$$

# The Dispatch Rule Revisited

$$
\frac{
\begin{array}{c}
O, M \vdash e_0 : T_0 \\
O, M \vdash e_1 : T_1 \\
\ldots \\
O, M \vdash e_n : T_n \\
M(T_0, f) = (T_1', \ldots T_n', T_{n+1}) \\
T_i \leq T_i' \;\; \text{for } 1 \leq i \leq n
\end{array}
}{
O, M \vdash e_0.f(e_1, \ldots, e_n) : T_{n+1}
} \quad \text{[Dispatch]}
$$

# Static Dispatch

- Static dispatch is a variation on normal dispatch
- The method is found in the class explicitly named by the programmer
- The inferred type of the dispatch expression must conform to the specified type

# Static Dispatch (Cont.)

$$\frac{\begin{array}{c} O, M \vdash e_0 : T_0 \\ O, M \vdash e_1 : T_1 \\ \ldots \\ O, M \vdash e_n : T_n \\ T_0 \leq T \\ M(T_0, f) = (T_{1'}, \ldots T_{n'}, T_{n+1}) \\ T_i \leq T_{i'} \;\; \text{for } 1 \leq i \leq n \end{array}}{O, M \vdash e_0@T.f(e1, \ldots, e_n) : T_{n+1}} \quad \text{[StaticDispatch]}$$

# The Method Environment

- The method environment must be added to all rules
- In most cases, M is passed down but not actually used
  - Only the dispatch rules use M

$$\frac{O,M \vdash e_1 : \text{Int} \quad O,M \vdash e_2 : \text{Int}}{O,M \vdash e_1 + e_2 : \text{Int}} \text{ [Add]}$$

# More Environments

- For some cases involving SELF_TYPE, we need to know the class in which an expression appears
- The full type environment for COOL:
  - A mapping O giving types to object id's
  - A mapping M giving types to methods
  - The current class C

# Sentences

The form of a *sentence* in the logic is

$$O,M,C \text{ ` } e: T$$

Example:

$$\frac{O,M,C \text{ ` } e_1 : \text{Int} \qquad O,M,C \text{ ` } e_2 : \text{Int}}{O,M,C \text{ ` } e_1 + e_2 : \text{Int}} \quad [\text{Add}]$$

# Type Systems

- The rules in this lecture are COOL-specific
  - More info on rules for self next time
  - Other languages have very different rules
- General themes
  - Type rules are defined on the structure of expressions
  - Types of variables are modeled by an environment
- Warning: Type rules are very compact!

# One-Pass Type Checking

- COOL type checking can be implemented in a single traversal over the AST
- Type environment is passed down the tree
  - From parent to child
- Types are passed up the tree
  - From child to parent

# Implementing Type Systems

$$\frac{O,M,C \vdash e_1 : \text{Int} \quad O,M,C \vdash e_2 : \text{Int}}{O,M,C \vdash e_1 + e_2 : \text{Int}} \quad \text{[Add]}$$

TypeCheck(Environment, $e_1$ + $e_2$) = {
  $T_1$ = TypeCheck(Environment, $e_1$);
  $T_2$ = TypeCheck(Environment, $e_2$);
  Check $T_1$ == $T_2$ == Int;
  return Int; }