Scoping and Type Checking

"Nobody understands me."
First, Written Assignments

- *Pick 'em up!* Even if you got a passing grade you’ll want to see what we marked up.
- Derivations and parse trees are closely related, but if we ask you to draw a parse tree you must *draw the parse tree*.
- WA2#4 was *in the book* (Fig 2.34; you just had to substitute in k=3):
Next. Semantic Fever: Catch it!
Course Goals and Objectives

• At the end of this course, you will be acquainted with the fundamental concepts in the design and implementation of high-level programming languages. In particular, you will understand the theory and practice of lexing, parsing, semantic analysis, and code interpretation. You will also have gained practical experience programming in multiple different languages.
In One Slide

- **Scoping rules** match identifier **uses** with identifier **definitions**.
- A **type** is a set of **values** coupled with a set of **operations** on those values.
- A **type system** specifies which operations are **valid** for which types.
- **Type checking** can be done **statically** (at compile time) or **dynamically** (at run time).
Lecture Outline

• The role of semantic analysis in a compiler
  - A laundry list of tasks
• Scope
• Types
The Interpreter/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 more errors
What’s Wrong?

- Example 1
  
  ```
  let y: Int in x + 3
  ```

- Example 2
  
  ```
  let y: String ← "abc" in y + 3
  ```
Why a Separate Semantic Analysis?

• Parsing cannot catch some errors

• Some language constructs are not context-free
  - Example: All used variables must have been declared (i.e. scoping)
  - Example: A method must be invoked with arguments of proper type (i.e. typing)
What Does Semantic Analysis Do?

• Many kinds of checks . . . cool checks:
  1. All identifiers are declared
  2. Static Types
  3. Inheritance relationships (no cycles, etc.)
  4. Classes defined only once
  5. Methods in a class defined only once
  6. Reserved identifiers are not misused
   And others . . .

• The requirements depend on the language
  - Which of these are checked by Ruby? Python?
Scope

• **Scoping rules** match identifier uses with identifier declarations
  - Important semantic analysis step in most languages
  - Including COOL!
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
  - Cool has static scope

• A few languages are dynamically scoped
  - Lisp, SNOBOL, Tex
  - Lisp has changed to mostly static scoping
  - Scope depends on execution of the program
Static Scoping Example

```plaintext
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;
    };
    x;
}
```

Uses of `x` refer to closest enclosing definition
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)
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 \( n \)
Implementing . . . (Cont.)

• Example: the scope of let bindings is one subtree

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

• \(x\) can be used in subtree \(e\)
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
  - You’ll need to make one for PA4
  - OCaml’s `Hashtbl` is designed to be a symbol table, so if you saved OCaml ... no, wait ...
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: Test in ... 
};

Class Test {
    ... 
};
More Scope in Cool

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

```kotlin
Class Foo {
    f(): Int { tm };  
    tm: Int ← 0;
}
```
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
  - This is standard inheritance!

- Methods may also be redefined (overridden)
Class Definitions

• Class names can be used before being defined
• We can’t check this property
  - using a symbol table
  - or even in one pass :-(

• Solution
  - Pass 1: Collect all class names
  - Pass 2: Do the checking
  - ?
  - Pass 4: Profit!

• Semantic analysis requires multiple passes
  - Probably more than two
Q: Advertising (832 / 842)

- Translate the last line in this French M&Ms jingle: Nous sommes les M&Ms / Nous sommes les M&Ms / Des belles couleurs en choix / Des belles couleurs en choix / Tout le monde nous aime / C'est nous, les M&Ms / M&Ms fondent dans la bouche, pas dans la main.
Real-World Languages

• This Asian language, sometimes called Siamese, is mutually intelligible with Lao and is spoken by 26+ million. It is tonal and has a complex writing system. The language's literature is influenced by India; its literature epic is a version of the Ramayana.

- Example: สัมผัสี
This line of female dolls with fruit-dessert names was initially introduced in 1980 and included sidekicks Blueberry Muffin and Crepe Suzette to help fight against Sour Grapes.
Types

• What is a type?
  - The notion varies from language to language

• Consensus
  - A set of values
  - A set of valid 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

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

What are the types of $r1, $r2, $r3?
Types and Operations

- Certain operations are **legal** or **valid** 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!
    - Our last, best hope ... for victory!

- Type systems provide a concise formalization of the semantic checking rules
What Can Types do For Us?

- Can detect certain kinds of errors
- Memory errors:
  - Reading from an invalid pointer, etc.
- Violation of **abstraction** boundaries:

```java
class FileSystem {
    open(x : String) : File {
        ...
    }
    ...
}

class Client {
    f(fs : FileSystem) {
        File fdesc <- fs.open("foo")
        ...
    } -- f cannot see inside fdesc!
}
```
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, OCaml, C#, C++, ...)
  
  - **Dynamically typed**: Almost all checking of types is done as part of program execution (Scheme, Ruby, Python, PHP, JavaScript, ...)
  
  - **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 easier in a dynamic type system
The Type Wars (Cont.)

• In practice, most code is written in statically typed languages with an “escape” mechanism
  - Unsafe casts in C, native methods in Java, unsafe modules in Modula-3

• Dynamic typing (sometimes called “duck typing”) is big in the scripting / glue world
Cool Types

• The **types** are:
  - Class names
  - SELF_TYPE

• There are *no* unboxed base types (**int** in Java)

• The user declares types for all 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 are often used interchangeably
Rules of Inference

• We have seen two examples of formal notation specifying parts of a compiler
  - Regular expressions (for the lexer)
  - Context-free grammars (for the parser)

• The appropriate formalism for type checking is logical rules of inference
Why Rules of Inference?

- **Inference rules** have the form
  
  \textit{If Hypothesis is true, then Conclusion is true}

- Type checking computes via reasoning
  
  \textit{If } E_1 \text{ and } E_2 \text{ have certain types, then } E_3 \text{ 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 $T$”
English to Inference Rules (2)

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

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

$$(e_1 : \text{Int} \land e_2 : \text{Int}) \Rightarrow e_1 + e_2 : \text{Int}$$
English to Inference Rules (3)

The statement

\[(e_1 : \text{Int} \land e_2 : \text{Int}) \Rightarrow e_1 + e_2 : \text{Int}\]

is a special case of

\[(\text{Hypothesis}_1 \land \ldots \land \text{Hypothesis}_n) \Rightarrow \text{Conclusion}\]

This is an inference rule
Notation for Inference Rules

• By tradition inference rules are written

\[ \vdash \text{Hypothesis}_1 \quad \ldots \quad \vdash \text{Hypothesis}_n \quad \vdash \text{Conclusion} \]

• Cool type rules have hypotheses and conclusions of the form:

\[ \vdash e : T \]

• \( \vdash \) means “we can prove that . . .”
Two Rules

\[ \vdash i : \text{Int} \quad (i \text{ is an integer}) \]

\[ \vdash e_1 : \text{Int} \]
\[ \vdash e_2 : \text{Int} \]
\[ \vdash e_1 + e_2 : \text{Int} \quad [\text{Add}] \]
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
• We can fill the template with ANY expression!

\[ \vdash \text{true} : \text{Int} \quad \vdash \text{false} : \text{Int} \]

\[ \vdash \text{true} + \text{false} : \text{Int} \]
Example: $1 + 2$

\[ \vdash 1 : \text{Int} \quad \vdash 2 : \text{Int} \]

\[ \vdash 1 + 2 : \text{Int} \]

All cats have four legs. I have four legs. Therefore, I am a cat.
Homework

- Compilers: PA6c Checkpoint Due
- Spring Break!
- Guest Lecture on Return: Claire Le Goues
- Should I put off PA4c?