1. Reading-in the Program: Lexing, Parsing, & Analysis
   - Regular expressions, scanner generators.
   - Syntax analysis, bottom-up parsing.
   - LR(0), LR(1) & LALR(1) parsing algorithm & parsing tables.
   - Classification of context free grammars and languages.
   - Error handling
   - Semantic analysis and Type checking.

2. Executing the Program: Code Generation
   - Generation of intermediate code
   - Generation of unoptimized code

3. Making the Program Run Fast: Code Optimization
   - Control-flow analysis
   - Data-flow analysis
   - Traditional Optimizations
   - Redundancy Elimination Optimizations
   - Loop Optimizations
   - Procedure Optimizations
   - Register Allocation
   - Instruction Scheduling
   - Instruction Optimizations

Overview of Semantic Analysis
Adapted from Lectures by
Profs. Alex Aiken and George Necula (UCB)
and Profs. Martin Rinard and Saman Amarasinghe (MIT)

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: 
      \( \{ \text{wcw | w \in (a + b)^*} \} \)
    - The 1st \(w\) represents a declaration; the 2nd \(w\) represents a use.
- Dynamic Check: 
  - Array bounds check
  - Null pointer dereference check
What Does Semantic Analysis Do?

Checks of many kinds . . .
cool checks:
1. All identifiers are declared.
2. Type compatibility.
3. Inheritance relationships (e.g., acyclic).
4. Classes defined only once.
5. Methods in a class defined only once.
6. Reserved identifiers are not misused.

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.
  - (Java) Local variables initialized before first use.
  - Each exit path returns a value of the correct type.
- (Java) Statement reachability check (Dead-code).

- Uniqueness Checks
  - No identifier can be used for two different definitions in the same scope.

(cont’d)

- Type checks
  - Number of arguments (in call) matches the number of formals (in declaration) 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".

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

Scope (Cont.)

- A declaration introduces an entity into a program and includes an identifier.

- The scope of a declaration is the portion of the program text in which the declared entity can be referred to using the identifier.
  - The same identifier may refer to different things in different parts of the program.
  - An identifier may have restricted scope.

Static vs. Dynamic Scope

- Most languages have static scope.
  - Scope depends only on the program text, not runtime 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.
Static Scoping Example

```
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;
  ```

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 in a class (introduce object ids)
  - Case expressions (introduce object ids)

Scope in Cool (Cont.)

- Not all kinds of identifiers follow the most-closely nested rule.
  - Cf. static vs lexical scoping
- 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.
  - No explicit forward declarations necessary.

Example: Use Before Definition

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

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

Class Foo {
  f(): Int { a);
  a: 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 may be defined in some parent class. (inheritance)

- Methods may also be redefined (overridden).
Scope in Java: Resolving Names

- Scopes can overlap.
- To disambiguate a name:
  - Use contextual information to determine if it refers to a package, a type, a method, a label, a variable (field/local/formal) etc.
  - Use signature to resolve method names.
  - Within nested scopes, type names & variable names resolved by hiding or overriding.
  - Otherwise, "Ambiguity error".
  - Possible remedy: use fully qualified name.

Meaning of a Name Context Dependent

```java
package Reuse;
class Reuse {
    Reuse(Reuse Reuse) {
        for (;;) {
            if (Reuse . Reuse (Reuse) == Reuse)
                break Reuse;
        }
        return Reuse;
    }
}
```

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.
  - Example: the scope of let bindings is subtree \( e \)
    ```java
    let x: Int ← 0 in e
    ```

Symbol Tables

- Consider again: let \( x: \text{Int} \leftarrow 0 \) 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.

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?

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);
  - Mutual Recursion
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 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.

Further Issues relevant to Symbol Tables

- Language Implementation Issues
  - Techniques for efficient access: Hash Tables
- Language Design Issues
  - Object-based languages (E.g., Ada, Modula-2)
    - Importing and exporting names
    - Static fields vs Instance fields
      - E.g., Interpreters developed in CS784
  - Object-oriented languages (E.g., C++, Java, Eiffel)
    - Name spaces/Environments: Class and Package hierarchy
    - Access control: private, protected, public, ...
      - E.g., Java Language Spec discussed in CS884

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

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

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

- 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.
    - Ensures 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
  - Example Program
  ```
  var: integer; var + 1023
  ```

Parser actions

- Example actions
  ```
  E → literal (E.type = char;)
  E → num     (E.type = integer;)
  ```

- If type checking is correct, then
  ```
  E → id     (E.type = lookup_type(id.name);)
  ```

- If no error, then
  ```
  E → E1 + E2 (if E1.type == integer and
                E2.type == integer then
                E.type = integer
                else
                E.type = type_error)
  ```
Parser actions

\[ E \rightarrow E_1 [E_2 ] \]

\{ if \( E_2 \) type == integer and \( E_1 \) type == array(s, t) then \( E \) type = s else \( E \) type = type_error \}