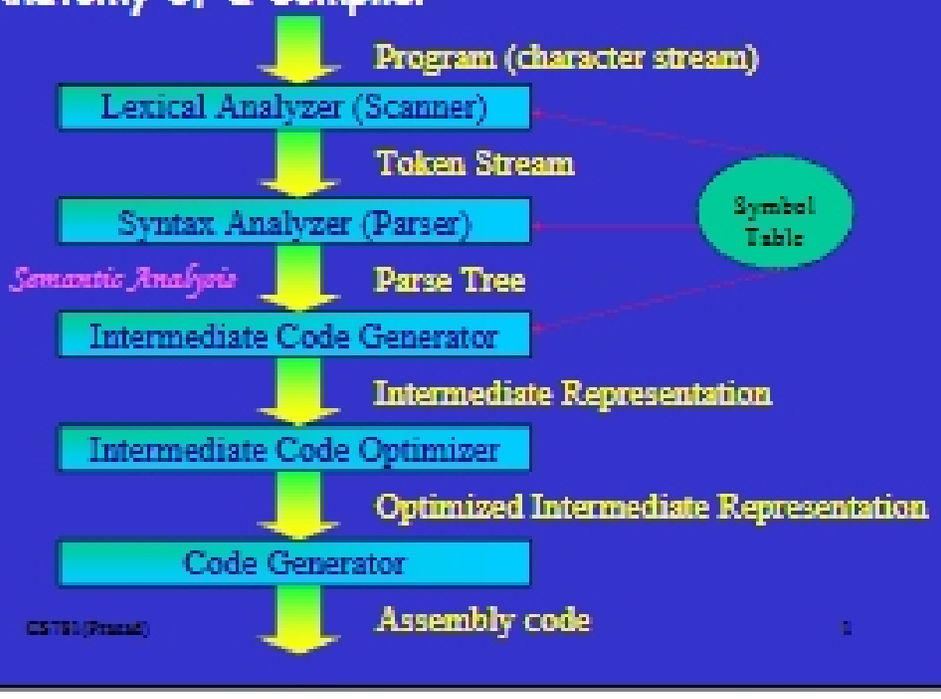
## Anatomy of a Compiler






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)*



### 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



### 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:
      $\{ wcw \mid w \in (a + b)^* \}$
    - The 1st $w$ represents a declaration; the 2nd $w$ represents a use.
- Dynamic Check:
  - Array bounds check
  - Null pointer dereference check

## What Does Semantic Analysis Do?

Checks of many kinds . . .

**coolc** 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 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.

## 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 id's)
  - Formal parameters (introduce object id's)
  - Attribute definitions in a class (introduce object id's)
  - Case expressions (introduce object id's)



## 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).