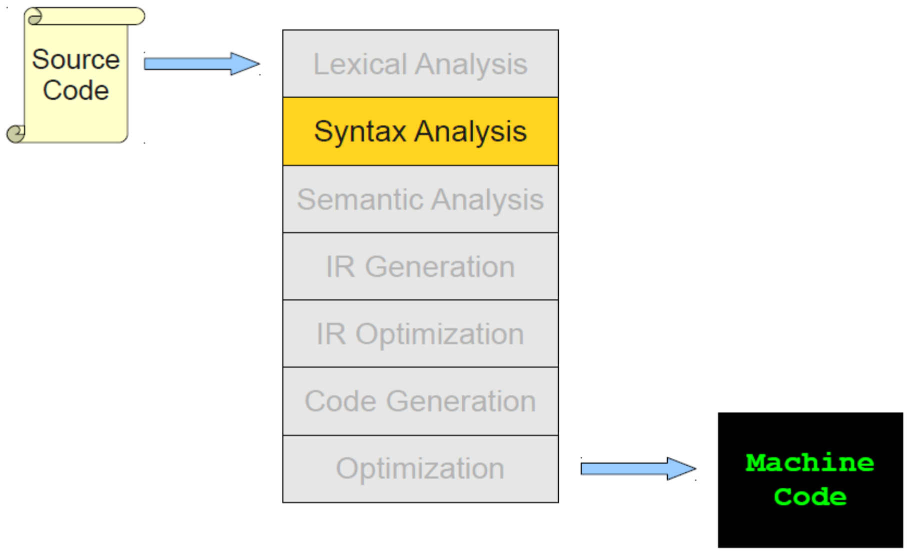
Semantic Analysis

Outline

- The role of semantic analysis in a compiler
  - A laundry list of tasks
- **Scope**
  - Static vs. Dynamic scoping
  - Implementation: symbol tables
- **Types**
  - Static analyses that detect type errors
  - Statically vs. Dynamically typed languages

Where we are



The Compiler Front-End

Lexical analysis: program is *lexically* well-formed
- Tokens are legal
  - e.g. identifiers have valid names, no stray characters, etc.
- Detects inputs with illegal tokens

Parsing: program is *syntactically* well-formed
- Declarations have correct structure, expressions are syntactically valid, etc.
- Detects inputs with ill-formed syntax

Semantic analysis:
- Last "front end" compilation phase
- Catches all remaining errors

Beyond Syntax Errors

- What's wrong with this C code? (Note: it parses correctly)

```
foo(int a, char * s){...}

int bar() {
  int f[3];
  int i, j, k;
  char q, *p;
  float k;
  foo(f[6], 10, j);
  break;
  i->val = 42;
  j = m + k;
  printf("%s,%s.\n",p,q);
  goto label42;
}
```

- Undeclared identifier
- Multiply declared identifier
- Index out of bounds
- Wrong number or types of arguments to function call
- Incompatible types for operation
- break statement outside switch/loop
- goto with no label

Why Have a Separate Semantic Analysis?

Parsing cannot catch some errors

Some language constructs are not context-free

- Example: Identifier declaration and use
- An abstract version of the problem is:

$$L = \{ wcw \mid w \in (a + b)^* \}$$

- The 1st w represents the identifier's declaration; the 2nd w represents a use of the identifier
- This language is not context-free

What Does Semantic Analysis Do?

Performs checks beyond syntax of many kinds ...

Examples:

1. All used identifiers are declared
2. Identifiers declared only once
3. Types
4. Procedures and functions defined only once
5. Procedures and functions used with the right number and type of arguments

And others . . .

The requirements depend on the language

What's Wrong?

Example 1

```
let string y ← "abc" in y + 42
```

Example 2

```
let integer y in x + 42
```

Semantic Processing: Syntax-Directed Translation

Basic idea: Associate information with language constructs by attaching *attributes* to the grammar symbols that represent these constructs

- Values for attributes are computed using semantic rules associated with grammar productions
- An attribute can represent anything (reasonable) that we choose; e.g. a string, number, type, etc.
- A parse tree showing the values of attributes at each node is called an *annotated parse tree*

Attributes of an Identifier

name: character string (obtained from scanner)

scope: program region in which identifier is valid

type:

- integer
- array:
  - number of dimensions
  - upper and lower bounds for each dimension
  - type of elements
- function:
  - number and type of parameters (in order)
  - type of returned value
  - size of stack frame

Scope

- The scope of an identifier (a binding of a name to the entity it names) is the textual part of the program in which the binding is active
- Scope matches identifier declarations with uses
  - Important static analysis step in most languages

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* (lexical) scope
  - Scope depends only on the physical structure of program text, not its run-time behavior
  - The determination of scope is made by the compiler
  - C, Java, ML have static scope; so do most languages
- 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 integer x ← 0 in
  {
     x;
     let integer x ← 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 integer a ← 42 in f(3);
f(x) = a;
```

- When invoking `g(54)` the result will be `42`

Static vs. Dynamic Scope

```
Program scopes (input, output);
var a: integer;
procedure first;
  begin  a := 1;  end;
procedure second;
  var a: integer;
  begin  first;  end;
begin
  a := 2;  second;  write(a);
end.
```

With static scope rules, it prints 1

With dynamic scope rules, it prints 2

Dynamic Scope (Cont.)

- With dynamic scope, bindings cannot always be resolved by examining the program because they are dependent on calling sequences
- Dynamic scope rules are usually encountered in interpreted languages
- Also, usually these languages do not normally have static type checking:
  - type determination is not always possible when dynamic rules are in effect

Scope of Identifiers

- In most programming languages identifier bindings are introduced by
  - Function declarations (introduce function names)
  - Procedure definitions (introduce procedure names)
  - Identifier declarations (introduce identifiers)
  - Formal parameters (introduce identifiers)

Scope of Identifiers (Cont.)

- Not all kinds of identifiers follow the most-closely nested scope rule
- For example, function declarations
  - often cannot be nested
  - are *globally visible* throughout the program
- In other words, a function name can be used before it is defined

Example: Use Before Definition

```
foo (integer x)
{
  integer y
  y ← bar(x)
  ...
}
bar (integer i): integer
{
  ...
}
```

Other Kinds of Scope

- In O-O languages, method and attribute names have more sophisticated (static) scope 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
  - 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

Implementing Most-Closely Nesting (Cont.)

- Example:
  - the scope of variable declarations is one subtree

```
let integer x ← 42 in E
```

  - **x** can be used in subtree **E**

Symbol Tables

Purpose: To hold information about identifiers that is computed at some point and looked up at later times during compilation

Examples:
- type of a variable
- entry point for a function

Operations: insert, lookup, delete

Common implementations: linked lists, hash tables

Symbol Tables

- Assuming static scope, consider again:

```
let integer x ← 42 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, if needed, 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 variable declarations
  - Symbols added one at a time
  - Declarations are perfectly nested

- Doesn't work for

```
foo(x: integer, x: float);
```

- Other problems?

A Fancier Symbol Table

- enter_scope() start/push 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() exits/pops the current scope

Function/Procedure Definitions

- Function names can be used prior to their definition
- We can't check that for function names
  - using a symbol table
  - or even in one pass
- Solution
  - Pass 1: Gather all function/procedure names
  - Pass 2: Do the checking
- Semantic analysis requires multiple passes
  - Probably more than two

Types

- What is a type?
  - This is a subject of some debate
  - The notion varies from language to language
- Consensus
  - A type is a set of values and
  - A set of operations on those values
- Type errors arise when operations are performed on values that do not support that operation

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 systems provide a concise formalization of the semantic checking rules

What Can Types do For Us?

- Allow for a more efficient compilation of programs
  - Allocate right amount of space for variables
    - Use fewer bits when possible
  - Select the right machine operations
- Detect statically certain kinds of errors
  - Memory errors
    - Reading from an invalid pointer, etc.
  - Violation of abstraction boundaries
  - Security and access rights violations

Type Checking Overview

Three kinds of languages:

Statically typed: All or almost all checking of types is done as part of compilation
- C, C++, ML, Haskell, Java, C#, ...

Dynamically typed: Almost all checking of types is done as part of program execution
- Scheme, Prolog, Erlang, Python, Ruby, PHP, Perl, ...

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, Java
- It is debatable whether this compromise represents the best or worst of both worlds