CSC 425 - Principles of Compiler Design I

Semantic Analysis

Outline

- The role of semantic analysis in a compiler
- Scope
  - static vs. dynamic scoping
  - implementation: symbol tables
- Types
  - static analyses that detect type errors
  - statically vs. dynamically typed languages

The Compiler Front-End

- Lexical analysis: the program is lexically well-formed
  - tokens are legal
  - detects inputs with illegal tokens
- Parsing
  - 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

- Example C program semantic errors:
  - Undeclared identifier
  - Multiple declarations of identifier
  - Index out of bounds
  - Incorrect number or types of arguments to function call
  - Incompatible types for operation
  - A `break` statement outside of a loop
  - A goto with no label

```
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;
}
```

Why Have a Separate Semantic Analysis Phase?

- 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 first w represents the identifier's declaration; the second 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:
  - All used identifiers are declared
  - Identifiers declared only once
  - Types
  - Procedures and functions defined only once
  - Procedures and functions used with the correct number and type of arguments
- The requirements depend on the language

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, for example, 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 the 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, an important static analysis step in most languages
- 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
- 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
- A few languages are dynamically scoped
  - Scope depends on execution of the program

Static Scoping Example

- Uses of `x` refer to the closest enclosing function

```
let integer x := 0 in
{
  x;
  let integer x := 1 in
    x;
  x;
}
```

Dynamic Scope

- A dynamically-scoped variable refers to the closest enclosing binding in the execution of the program
- Example: when invoking `g(54)` the result will be `42`

```
g(y) = let integer a := 42 in f(3);
f(x) = a;
```

Static vs. Dynamic Scope

- Example

```
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, the result is 1
- With dynamic scope, the result is 2

Dynamic Scope Continued

- 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
- Usually these languages to not normally have static type checking

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

- 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 object-oriented languages, method and attribute names have more sophisticated (static) scope rules
- A method may 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 node *n*
- When preforming semantic analysis on a portion of the AST, we need to know which identifiers are defined.

Implementing the Most Closely Nested Rule

- 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
- Example information:
  - 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 a definition of `x` to the current definitions, overriding any other definition of `x`
  - after processing `E`, remove the 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

- The 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 the top for `x` and return the first occurrence of `x` found or null if not found
  - `remove_symbol()` pop stack
- Why does this work?

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` is defined in the current scope
- `exit_scope()` exit/pop the current scope

Function/Procedure Definitions

- Function names can be used prior to their definition
- We cannot check that for function names
  - using a symbol table
  - or even using one pass
- Solution
  - pass 1: gather all function/procedure names
  - pass 2: do the checking
- Semantic analysis requires multiple passes

Types

- What is a type?
  - This is the 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

Types and Operations

- Consider the assembly language fragment

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

  What are the types of `$r1`, `$r2`, and `$r3`?
- Certain operations are legal for values of each type
  - It does not 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
- 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 the correct amount of space for variables
  - Select the correct machine instructions
- Statically detect 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
  - Dynamically typed: almost all checking of types is done as part of program execution
  - Untyped: no 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 protoyping is easier in a dynamic type system