Basic Semantics

CS152

Chris Pollett

Oct. 1, 2008.

Outline

- Attributes, Binding, and Semantic Functions
- Declarations, Blocks, and Scope

Semantics

- Recall a programming language is supposed to give us a way to write programs in a human readable and computer understandable format.
- So far we figured out how to specify the syntax of programming languages which are human understandable.
- We also have figured out how to write programs which can determine if an input file is a valid program in programming language.
- We next need to describe how we can go from parsing to having machine code which could execute on some computer.
- That is the subject of semantics.

Attributes

- Programming languages use **names/identifiers** to refer to various languages entities or constructs.
- To give meaning to these names it useful to use the concept of **location** of where the name refers to is stored and to use the concept of **value** for what is stored there.
- The meaning of a name is determined by the properties (aka **attributes**) associated with the name.
- For example, the declaration:

  const int n=5;

  Makes n into an inter constant with value 5. So the meaning of n is a name for a datatype attribute "integer constant" and a value attribute 5.

- The function declaration: double f(int n) { … }

  Associates the attribute function to the name f together with (a) the number, names, and data types of its parameters, (b) the datatype of the return value, and (c) the body of code to be executed.

Binding

- The process of associating an attribute to a name is called **binding**.
- Binding can occur in other places then in declarations. For example, x = 5 binds the value attribute 5 to the name x.
- One can classify attributes according to when in the translation/execution process it is computed and bound to a name. This is called **binding time**.
- **Static binding** occurs prior to execution. An example of this might be a declaration like: int x; //The binding of the datatype attribute to the name x occurs in the compiler.
- **Dynamic binding** occurs during execution. For example, a statement like x = 2; binds x to the value 2 at that place during execution. A declaration like y = new int; in C++ binds the value of the pointer y to a new storage location as that point in the code execution.

The Basic Semantic Function

- Bindings must be maintained by the translator so that appropriate meanings are given to names during translation and execution.
- We can think of the data structure that the translator uses as a functions that expresses the binding of attributes to names.
- This function which plays a fundamental role in language semantics is usually called the **symbol table**.

  SymbolTable: Names --> Attributes.
- This data structure typically changes as execution progresses as bindings are added, deleted, or updated.

Semantic Functions in Compilers and Interpreters

- A **compiler** typically only can compute static attributes in its symbol table.
- Runtime location and value changes are handled by the compiler generating code to maintain these values during execution.
- The memory allocation for this process is usually considered separately from the symbol table and is called the **environment**.
- Finally, the bindings of storage locations to values is called the **memory**.
- So for a compiler we have the three semantic functions:
  1. SymbolTable: Names --> Static Attributes.
  2. Environment: Names --> Locations .
  3. Memory: Locations --> Values .
- In an **interpreter** the symbol table and environment are combined and both static and dynamic attributes are computed during execution. The whole biding function is usually just called the **environment**.

Declarations

- Declarations are one of the main ways to create new bindings.
- Consider: int x;
- The data type of x is **explicitly** declared. However, the exact location where x is stored is bound only **implicitly** during execution and may depend on where this declaration occurred in the program.
- Some languages allow you to even have implicit declarations. For example, in PHP a variable gets implicitly declared when you first assign it. $x=6.5; //implicitly creates $x and makes it a double.
- Sometimes languages have a distinction between bindings which bind all potential attributes called **definitions** versus bindings which bind only some attributes which are called **declarations**.
- For example, a C prototype void f(int a); is a declaration but not a definition; whereas, the actual code of f would be its definition.

Blocks

- One language construct which is often associated with declarations is the **block.**
- A block consists of a sequence of declarations followed by statements, then surrounded by syntactic markers such as braces or begin-end pairs.
- In C, blocks are called compound statements. They can appear as the body of a function or can appear anywhere an ordinary program statement could appear:

  void p() {double r,z; … {int x, y; /* nested block/*…}}

- In addition to declarations associated with blocks C also has an external or global set of declarations outside any compound statement:

  int x;

  void main() {/* some code*/}

More on Blocks

- Declarations that are associated with the specific current block are called **local**. Otherwise, the declaration is called **nonlocal**.
- Algol60 was the first language to associate declarations with blocks and to support nesting of blocks.
- All Algol descendants exhibit this **block structure** in various ways.
- For example, in Ada a block may be written using begin-end pairs:

```
declare x: integer;
            y: boolean;
begin
    x:= 2;
    y:= true;
end;
```

Other Language Constructs Supporting Declarations

- All structured data types are defined using local declarations associated with the type.
- For example, in C:

```
struct A
{ int x; double y;
   struct {int* x; char y;} z;
};
```

- Similarly, object-oriented languages such as Java or Smalltalk use the notion of **class** as an important source of declarations.
- Declarations can even be collected into larger groupings in some languages such as packages in Ada or Java; namespaces in C++; or modules in ML or Haskell.

Scope

- Each binding associated with a declaration has an attribute that is determined by the position of the binding within the program and by the language rules for the binding.
- The **scope of the binding** is the region of the program over which the binding is maintained.
- Usually, people refer to the scope of a binding rather than of a name to avoid confusions with code like:

```
void p() {int x; …}
void q() {char x;}
```

- For a block structured language one typically has a **lexical scope** rule. That is, the scope of binding is limited to the block in which it appears.