Lecture 4

Introduction to Semantics

The semantics of a language is the set MEANINGS or ATTRIBUTES for statements and programs.

There are rules and meanings in programs that can't be easily identified by syntax rules using BNF or EBNF.

EXAMPLE: In Ada the name of a program (that following the keyword "procedure" but be repeated at the end of the program after the end statements.


Semantic Rule: <prog-name>[1].string SAME AS <prog-name>[2].string

Look at an Operational Semantic (how a program executes) problem.

A C for loop looks like for (expression1; expression2; expression3) {
    {
        stuff;
    }
}

Translated to more primitive, virtual machine code, it may look like

expression1;
loop: if expression2 == 0 goto out;
    stuff;
expression3;
    goto loop;
out:

More on Attributes

Pascal: const k = 10; k is name attribute; other attributes: "const" "integer value 10"

C: void f ( int n ) { body } f is name attribute; other attributes: 1) function, 2) number, names, types of parameters; 3 ) body of function

C: int * p; p = new int; location attribute

Classifying attributes at compile time is called static binding
Otherwise classifying attributes is called dynamic binding
**Name** (identifiers): case sensitive (often); keywords: reserved words and pre-defined words

**Variables** have attributes: a) name b) address c) value d) type e) lifetime and f) scope

- **Address**: called l-value (left value of an assignment)

- **Aliases**: two names for the same variable location (e.g. pointers and reference calls)
  Side effects of aliases

- **Value**: r-value (right value of an assignment)
  Note: named constants and variable assignments

**Type**

- Type binding:
  Some languages assume type bindings FORTRAN, BASIC
  Dynamic type binding: changing variable types at runtime

- **Lifetime**: Time when a variable is bound to a memory location.
  Note: Static variables

---

**Stack-dynamic variables**: storage bindings are created when their declaration statements are elaborated; for example function local variables.

**Explicit heap-dynamic variables**: new/delete in C++; all non-primitive objects in Java

**Implicit heap-dynamic variables**: a generic method of binding variables to memory locations whenever the variables are assigned values. (Arrays and strings in Perl and Javascript).

Type checking; type coercion (automatic conversion)

**Strong typing languages**
- FORTRAN not
- Pascal nearly (variant record -- an exception)
- Ada quite nearly (UNCHECKED_CONVERSION)
- C, C++ not
- Java quite nearly

Type Compatibility

**Name type compatibility**: two variables have compatible types only if they are in either the same declaration or in declarations that use the same type name.

**Structure type compatibility**: two variables have compatible types if their types have identical structures.
Examples:
Pascal

```pascal
    type
        Celsius = integer;
        Fahrenheit = integer;
    var
        C : Celsius;  F : Fahrenheit;  (structure compatible -- not name compatible)
        T : Celsius  (C and T have name compatibility)
```

Note: C/C++ typedef defines a name -- not a new type

Ada derived types:
```
    type CELSIUS is new FLOAT;
    type FAHRENHEIT is new FLOAT;
    C : CELSIUS;  F : FAHRENHEIT; --- no compatibility
```

Scope

*The scope of a variable is the portion of the code where the variable can be referenced.*

Blocked Code:

Ada:
```
    declare ..... 
    begin
        ...
        ...
    end;
```

Pascal: begin end

C++: { }

Special note: in C++ variable definitions may occur anywhere in a function.

**Static Scoping**: the scope of a variable can be statically determined -- prior to execution at compile time.

Generally, in static scoping a variable can be referenced anywhere in the block it was defined. With special exceptions if a "more local" variable is declared with the same name, the variable with the greater scope can't be referenced until the end of the block of the second variable (with the same name). Some languages allow special access to the variable with greater scope.

Static Scoping has its problems. Failure to localize data; side effects, and so forth. Solutions: localization of data and data encapsulation.
**Dynamic Scoping** (determining the scope at run time by the order of variable references).

*Example (from Sabesta)*

```plaintext
procedure big;
  var x : integer;

  procedure sub1;
  begin
    ...x...
  end;

  procedure sub2;
  var x : integer
  begin
    ...
  end
begin
  ...
end;
```

Assume here that dynamic scoping rules apply to non-local references. The meaning of the identifier `x` referenced in `sub1` is dynamic -- it cannot be determined at compile time. It may reference the variable from either declaration of `x`, depending on the calling sequence.

Suppose that `big` calls `sub2` which calls `sub1`. The reference to `x` in `sub1` refers to the `x` declared in `sub2`. We say "the dynamic parent of `sub1` is `sub2".

Suppose that `big` calls `sub1`. In this case the dynamic parent of `sub1` is `big`. The reference is to the `x` declared in `big`.

**Scope vs. Lifetime**

Remember that scope really refers to where the variable is accessible. Lifetime refers to when the variable is assigned a memory location. Examples. The lifetime of a global variable in C is until the end of the program; however, local variables with the same name may temporarily cut its scope. Static variables in C are also assigned to memory locations when they are first declared, but the scopes are local to the functions in which they are declared. A local variable that calls a function with no parameter reference call to that variable has its scope temporarily isolated from being accessed.

Example:
void doit();

int main ()
{
    int x;
    stuff;
    doit();
    morestuff;

    return 0;
}

void doit()
{
    int x;
    stuff3;
}

The life time of the x in main is until the end of the program; however, the scope of x is to the end of the program EXCEPT when doit() is executing. The scope and lifetime of the x in doit(); is until the end of the function.