Semantics

Recall :

> Programming Language syntax means what the language constructs look like.

Also recall :

> Programming Language semantics means what those language constructs actually do(meaning).

Programming language semantics are much more complex to express than the syntax. Programming language semantics can be specified by :

1. The Programming language reference manual (most common and simple).
2. Translator (Compiler or Interpreter).
    - By Experiment.
    - Execute programs to find out what they do.
    - Machine dependant(generally it is not portable).
3. Formal Definition (mathematical model). It is complex and abstract.

We will mainly be using the first method. We will also use ALGOL-like languages in our discussion.

Binding

Using names or identifiers in a programming language is a basic, fundamental abstraction - variable names, constant names, procedure and function names are all examples of this.

Related to names is the concept of **location**. Simply put, the location is the address of the name in memory. Another thing related to the name is the **value**, which is the storable quantity in memory.

But how is the meaning of names determined?

It is determined by its **attributes** (properties associated with it).

For example :

```
const n = 15;
```

in this declaration, we associated 2 attributes :

1. It is a constant name.
2. it has a value of 15.

Another example :

```
VAR
  x:integer;
```

again, 2 attributes are associated with this name :

1. It is a variable.
2. It is an integer.

Another example :

```
function compute(n:integer, x:Real):Real;
    Begin
    .
    .
    .
    .
    .
    end;
```

Associated with the name `compute` (function name) is :

1. It's type : a function name.
2. Number and type of parameters : it takes 2 parameters, one of type `integer`. and one of type `Real`.
3. It's return value : The function returns `Real`.
4. The code body of the function.

Another example :

```
Var
  y:^integer;
```

Associated with the name `y` is :

1. It's a variable name.
2. It's a pointer variable to an integer.

Notice that in all the examples above are attributes that are determined at declaration. However, we can assign attributes outside the declaration. For example :

```
x := 2
```

this means that we add a new attribute to the name `x`, which is the value.

In the example :

```
Var
  y:^integer
```

we can say

```
new(y);
```

in this case, we add a third attribute to `y` which is the location.

When we first declared y, it pointed to junk (something random). When we used `new(y)`, pascal reserved a place in the memory the size of an integer and changed the reference to it (without having to name it, unlike C).

The process of associating attributes to names is called **Binding**. This happens at **Binding Time**.

> Binding Time : The time during the translation(compilation) process when the attribute is computed and associated to the name.

There are 3 kinds of binding times.

1. Static Binding : binding which occurs before execution. We call those attributes static attributes. An example of a language that uses this is FORTRAN.
2. Dynamic Binding : binding which occurs during execution. An example of a language that uses this is LISP.
3. Mixed Binding : A mix of both approaches. Some attributes are bound dynamically, while some are bound statically. An example of a language that uses this is C.

Examples :

1. `const n=2` is a static attribute. This is because the attributes `constant` and `value=2` is assigned during compilation.
2. In `x:integer` the attributes `variable` and `type integer` are also static attributes. However, when we say `x:=2`, the attribute `value=2` is a dynamic attribute because it is assigned during execution.
3. In `y^:integer`, the attributes `variable`, `integer`, and `type pointer` are static attributes. However, in the statement `new(y)`, the attribute `location` is a dynamic attribute.

Binding can be performed prior to translation. There are a number of examples of this :

- Binding values to the `integer` type or the `boolean` type in PASCAL is performed at **language definition time** . Meaning that if the `integer` type is 16 bits, then its range would be (-2^16) ≤ integer ≤ (2^16 - 1).
  - This is the same for the `TRUE` and `FALSE` values binded to `boolean` type.
- The constant `MAXINT` in PASCAL is defined at **implementation time**(when the language was created). it is an `integer` with the highest possible value for the platform.

In short, Binding can be performed at :

- Language definition time.
- Language implementation time.
- Translation time.--> this is lexical scoping-->most languages are lexically scoping.
  - at lexical analysis.
  - at syntax analysis.
  - at code generation.
  - However, this binding is static.
  - Variables can only be called in the block they are defined in.
- Execution time.--> languages that use this are usually interpreters and functional languages.
  - This binding is dynamic.
  - Variables can be called globaly, which might cause issues.

Symbol Table

The Symbol Table is a special data structure used to maintain the binding during the translation process.

Memory

Memory is the binding the storage locations to values.

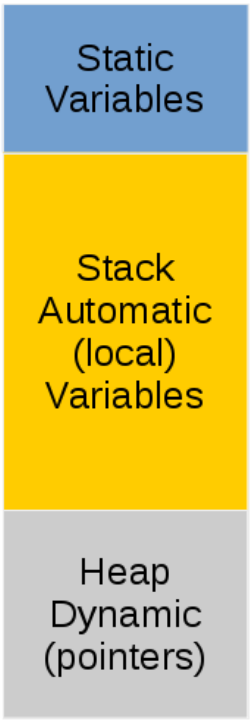
Environment

The Environment is the memory allocation part of the execution process. ie, binding names to the storage locations is called Environment.

There are three kinds of allocations in the environment.

1. Static --> global variables.
2. Automatic --> local variables(stack).
3. Dynamic --> pointers.

Generally, in block-structued languages, when the process is created, the environment is allocated like this:



This allows room for growing and shrinking. This happens when functions are called, and their variables are allocated while the program is running, and discarded after. Each call is called an **activation**.

Declarations and Blocks

Declarations are the principle method to establish binding. There are 2 types of declarations:

1. Explicit Declaration:
   - Pascal :

     ```
     var
     x:integer
     ok,y,Boolean;
     ```

   - ALGOL68 :

     ```
     Begin
     Integer X;
     Boolean ok;
     End
     ```

   - ADA :

     ```
     Declare
     x:integer;
     y:boolean;
     ```

   - C :

     ```
     int n;
     ```

2. Implicit Declaration : The variable is declared when it is used. for example, `int n = 10`.

Declarations are associated with **blocks**. There are 2 types of blocks:

1. Main Program Block.

2. Procedure and Function Block.

for example, in PASCAL :

```
Program Test;
  Var
    .
    .
    .
  Procedure P;
    Var
      .
      .
      .
   Begin
    .
    .
    .
   End;

  function q:integer;
    Var
      .
      .
      .
   Begin
    .
    .
    .
   End;

  Begin(*main*)
    .
    .
    .
    .
  End.
```

These are all declarations for program Test. Regular declaration scoping applies here.

In ALGOL :

```
Begin
  Integer X;
  Boolean Y;
  .
  .
  .
  X := 2;
  Y ;= True
  .
  .
  .
End
```

In ADA:

```
Declare
  X : Integer;
  Y : Boolean;
  Begin
    X := 2;
    Y := 0;
  End;
```

Declarations bind different attributes to names especially the static type of attributes. Note that the declaration itself has an attribute, which is the position of the declaration in the program. This is important to determine the **scope/visibility** of the variable.

Scope of Declaration

The scope of declaration is :

> The region of the program over which the declaration covers. In block structured languages, such as PASCAL , the scope of declaration is limited to the block in which is declared/appears and all other nested blocks. Contained within this block.

In fact, a language like PASCAL has the following scope rule :

> The scope of declaration extends from the point it is declared to the end of the block.

for example :

```
Program scope;
  VAR X : Integer; ________
  Procedure P;     ______  |
    VAR X:Real;    ____  | |
    BEGIN             | | |
      .               | | |
      .               | | |
      .               | | |
    END;           ____| | |
  Procedure q;     ____  | |
    VAR Z:Boolean; __  | | |
    BEGIN            | | | |
      .              | | | |
      .              | | | |
      .              | | | |
    End;           __| | | |
                      | | |
  BEGIN(*main*)       | | |
    .                 | | |
    .                 | | |
    .                 | | |
    .                 | | |
  END.                | | |
                    __|_|_|
```

In ALGOL 60:

```
A:BEGIN
  Integer:X;
  Boolean:Y;
  X:=2;
    .
    .
    .
  B:BEGIN
     Integer c,d;
      .
      .
      .
    END
    .
    .
    .
   END
```

That is , `X` and `Y` have scope in blocks `A` and `B` , while `c` and `d` have scope only in `B` .

As we have seen before, declarations bind different attributes to names especially static attributes.

The declaration itself has an attribute which is the position or the location of declaration in the program.

An Important thing to note. The declarations in nested blocks takes precedence over previous declarations. A global variable `x` cannot be accessed in a block `B` that has a local variable `x`. We say we have a **scope hole** in `B`. That is why we differentiate between scope and **visibility**. Only the area where the declaration applies. So the scope is the visibility - holes.

Symbol Table (continued)

All the declarations and binding are established by a structure called the symbol table. In addition, the symbol table must maintain the scope of declaration. Different data structures can be used in the symbol table :

1. Hash Table --> static.
2. Linked List --> dynamic.
3. Tree Structure --> dynamic.

To maintain the scope of declarations correctly, the declarations should be processed using the stack concept(FILO). When entering a block, declarations are processed and attributes are added/binded to the symbol table(pushed to stack). When exiting from the block, the binding(the attributes) provided in the block are removed/popped from the stack.

Think of the symbol table as a set of names, each of which has a stack of declarations associated with it. The top of the stack is the current active declaration.

For example, consider the following pascal program :

```
Program Sym-Table;
VAR X:Integer;
    Y:Boolean;
Procedure P;
    VAR X:Bolean;
      .
      .
      .
    Procedure Q;
        VAR Y:integer
        Begin(*Q*)
          .
          .
          .
        END;
    Begin(*P*)
      .
      .
      .
    END;
Begin(*main*)
  .
  .
  .
  .
END.
```

Syntax Directed Translation

Lexical Structure --> Systematic algorithms exist --> Finite State Automata.

Syntax Structure --> Systematic algorithms exist --> Push Down Automata.

Semantic Structure --> Unfortunately, no systematic algorithm.

However, there is a framework for **intermediate code generation**, which is an extension of the context-free grammar called **syntax-directed translation**.

In syntax-directed translation, the algorithm allows what is called a **semantic action**, which is simply a subroutine(procedure/function) attached to some of the production rules of the context-free grammar.

A semantic action or a semantic rule is simply an output action added(associated) to the production rule of the grammar. For example, given the production

> A --> α

the semantic action is simply

> A --> α #β

where β is the semantic action.

Take the production

> A --> XYZ #α

assume that α is the semantic action/rule, then in the syntax-directed translation scheme, the semantic action α is called/executed whenever the parser recognizes or accepts a sentence w derived from A in top-down parsing.

> A --> XYZ -*-> w ∈ L(G)

In bottom-up parsers, the semantic action α is called whenever XYZ is reduced.

Generally, compilers generate/translate source code into another format which is easier for the compiler to understand(evaluate). This code is called **intermediate code**. There are different kinds of intermediate code :

1. Postfix Code : for example,

   `A + B * ( C - D ) + E`

   becomes

   `A B C D - \* + E +`

   in postfix.

   Another example :

   ```
   if a
       x
   else
       y
   ```

   would become

   `a x y ?`

   in postfix.

   Or another example :

   ```
   if a
       if(c - D)
           a + c
       else
           a * c
   else
       a + b
   a
   ```

   becomes

   `a c D - a c + a c * ? a b + ?`

   in postfix.

2. Three Address Code (TAC) : Each instruction has at most 3 components.

   for example :

`- w * x + ( y + z )`

would be

```
(1) -,w
(2) *,(1),x
(3) +,y,z
(4) +,(2),(3)
```

in TAC.

Another Example :

```
if (x > y)
    z = x
else
    z = y + 1
```

would be

```
(1) -,x,y
(2) JGZ,(1),(6)
(3) +,y,1
(4) =,z,(3)
(5) JMP,(7)
(6) =,z,x
(7) ......
```

in TAC.

3. Quadruples : Another form of intermediate code, which has at most 4 components. for example :

   `- w * x + ( y + z )`

   would be :

operation	operand 1	operand 2	result
-	w	_	R1
*	R1	x	R2
+	y	z	R3
+	R2	R3	R4

Semantic Rules

Simply put :

> Context Free Grammar + Semantic Rules = Syntax Directed Definitions

A Compiler has to do more than just recognise if a sequence of characters forms a valid sentence in the language. It must do something useful with the parsed sentence. The semantic actions of a parser perform useful operations.

- Error Checking, which includes :
  - Type mismatch
  - Undeclared variable
  - Reserved identifier misuse.
  - Multiple declaration of variable in a scope.
  - Accessing an out of scope variable.
  - Actual and formal parameter mismatch.
- Evaluation in the case of an interpreter.
- Begin the translation process in the case of a compiler.

Semantic Rules are actions that are performed when certain productions are found to be correctly derived(or reduced). A simple example would be a calculator that has this simple grammar with the attached semantic rules:

> E → E + T { E.value = E.value + T.value }

What this says is to add E to T and copy it to E whenever a sentence of the form

> E → E + T

is derived.

For a more detailed explanation about semantic rules, look here.