14 Types and Type Checking

Wednesday, April 3, 2024 12:31 PM

SEMANTICS

The study of meaning.

e.g. C++

```
float x, y, z;
*x = y[z] << x(z[3.14], x->z);
```

follows the syntax rules
but
it is meaningless.
becouse
operators don't match operands.

in programming languages, semantics are mostly about identifiers and their correct use.

- TYPE CHECKING:

¿Are named entities used in a matter consistent with their definition?

i.e.:
- using a variable with appropiate operators
- calling a function with appropiate number and types of parameters
- members of compound types are used correctly

¿How?

- The compiler/parser needs to remember the declaration of named entities and their definitions

- THE SYMBOL TABLE

A table of named entities and their attributes

names	attributes
x	int.

example of symbol table entries:

variable:

float x; x : float.

float x; x : float.

constant:

const int y=3 y : int, const, 3

type:

```
struct cell {
    int row, col
}
```

cell : struct, 2, int row, int col.

function:

```
void foo(int r, int c)
{ ≡
}
```

foo : function, void, 2, int r, int c

Entries are different in different prog. languages

C++

```
int z[3];
float foo ( int x, bool y, char& c);
```

name	attributes
z	int array.
foo(int, bool, char&)	function, 3, int, bool, char&, float

Python.

```
z = 3
def foo ( x, y, c) : .........
```

name	attribute
z	3
foo	3

Pascal

```
z : ARRAY [10..15] OF INTEGER;
FUNCTION bar ( x : INTEGER, VAR y : STRING ) : REAL
```

name	attributes
z	Array INT, 10..15
bar (INT, STRING) :REAL	function 2 INTEGER VAR STRING

Consequences:

```
int foo ( string y )…

int foo ( char* y ) …

string& foo ( string y ) ….
```
⊗

- TYPE CONVERSION

Some languages will automatically convert types when allowed.

e.g. a + x   a: float
x: int.

most parsers will identify the need of conversion and automatically convert.

int → float.

What converions are Implicit?

C++ : flexible.
Python : runs and maybe crash
Pascal : Strict.

E.g. C++

```
foo( Dog d )
{ ≡
}

foo( 3 );
```

```
class Dog:
{ ≡
  Dog (int n)
  { ≡
  }
}
```

becomes an Implicit conversion.

E.g C

```
int x = 70;
char c = '!';

while ( 'W' - x ) {
  c = '!' + x;
  printf("%c", c);
  x = x + 1;
}
```

('W' : char, x : int, 'W' - x : int; c : char, '!' : char, x : int)

{ 0 is false
{ everything else is true

ghi

Pascal

```
VAR X : INTEGER = 70;  c : CHAR;

WHILE (ORD('W') - x) > 0  DO BEGIN

  c := CHR( ORD('!') + x );

END;
```

(ORD : explicit; ORD('W') - x : int; (ORD('W') - x) > 0 : boolean. ORD('!') : char to int; ORD('!') + x : int; CHR(...) : int to char.)

- TYPE EQUIVALENCE

a = b

¿When are two entities of the same type?

1) Name type equivalence
   - two entities are of the same type if they are declared using the same type name.

Dog x; Cat y; Dog z;

just look at text

2) Structural type equivalence

- two entities are of the same type if they have the same internal structure.

Imagine.

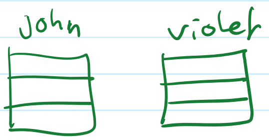
C's:

```
struct pnt        struct color      struct cell
{                 {                 {
  int x,y,z;        int r,g,b;        int x,y;
}                 }                 }
```

```
struct pnt john;
struct color violet;
struct cell bob;

john = violet; ✓

bob = john;
```



bob and john are not the same type.

- Hard. – you needs to test structure
  – structure could be nested.

- STRONG vs WEAK TYPING:

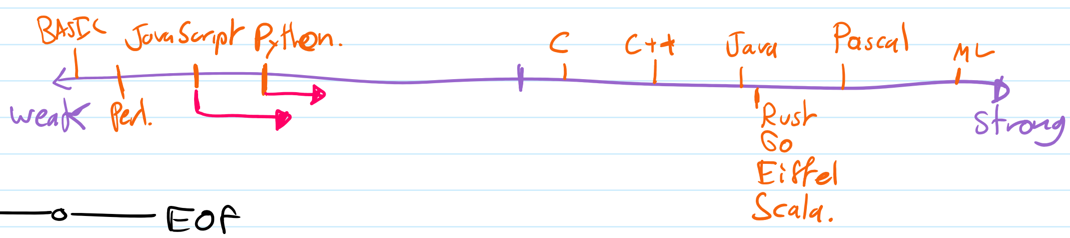
A characterization of Programming Languages.

Strongly typed if:

- type violations are detected at compile/parse time
- type conversions are explicit
- the type of a named entity remains fixed.

General idea behind Strong types:

Detect errors at compile time, instead of letting them happen at runtime.



——o—— EOF