Expressions and assignments

Contents

Arithmetic expressions

Operator precedence or strength: order of evaluation of expressions with more than one operator. (Parentheses can always be used to specify order explicitly.)

Operators are usually grouped:

- exponentiation (`**`)
- unary operators (`abs`, unary `+` and `-` etc.)
  (`not` is a unary operator, too)
- multiplicative (`*`, `/`, `div`, `mod` etc.)
  (`and` is multiplicative, too)
- additive (binary `+`, binary `-` etc.)
  (`or` is additive , too)

Rules are confusing and widely different.

Pascal: multiplicative > additive

C: self-increment, unary > multiplicative > additive

Ada: ** > multiplicative > unary +, - > additive

Fortran: ** > multiplicative > additive

Associativity

Let ¥ be any binary operator.

Left to right: `x ¥ y ¥ z = (x ¥ y) ¥ z`

Pascal, Ada, C (all normal operators)

Right to left: `x ¥ y ¥ z = x ¥ (y ¥ z)`

C (self-increment operators `++` and `--`)

Nonassociative (exponentiation in Ada):

`x**y**z` is syntactically incorrect, though `(x**y)**z` and `x**(y**z)` are OK.

No precedence, one associativity rule

This is what we have in APL: always right to left.

```
x + y * z means x + (y * z)
x * y + z means x * (y + z)   (!?)
```

In Smalltalk: always left to right.

```
x + y * z means (x + y) * z   (!?)
x * y + z means (x * y) + z
```

Evaluation of arguments, side-effects

A function that appears in an expression may have a side-effect (change to some non-local object, not mentioned in the expression). Example:

```
function twice(var x: real): real;
begin x := x + x; twice := x end;
```

In the statement

```
z := twice(y);
```

the value of `y` is changed "secretly".

Such effects are to be avoided, if possible. Built-in functions seldom have side-effects, arithmetic functions—never!

Conditional expressions

In Algol 60:

```
if x > 0 then 1 else
if x = 0 then 0 else -1
```

The same in C:

```
(x > 0 ? 1 : (x == 0 ? 0 : -1))
```


Overloading

One name or symbol—more than one distinct use.

Examples in Pascal:

`+`
: integer addition, floating-point addition, string concatenation, set union

`*`
: integer and floating-point multiplication, set intersection

`abs`
: integer → integer, real → real

`mod`, `div`
: no overloading

Overloading can be always resolved by context (all operands have known types):

`2 + 3`: integers, `['a']+['c','d']`: sets

In Ada, overloading is an important element of design. Ada is extendible: a new meaning can be given to an operator in addition to what it already means. Overloading is also possible in C++.


```
type date is
record
  day: 1..31;
  month: 1..12;
  year: 1000..9999;
end record;

DT: date;

function "+"(D: date; I: integer)
            return date is
NewD: date;
begin
  -- code that assigns D + I days
  -- to NewD;
  return NewD;
end;
```

Now, we can use this operator as follows:

```
DT := (18, 10, 1994) + 2;
```

Ada also has only two standard I/O procedure names: `get`, `put`—both heavily overloaded.

Overloading may be quite confusing. In C:

& means "bitwise conjunction" and "address of",

* means "multiplication" and "dereference".

In PL/I, = means "equality" and "assignment".

In Prolog, the comma is overloaded in a rather careless manner.

This comma can be read "and":

```
a :- b, c, d.
```

This comma separates arguments:

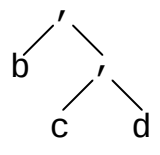
```
a(b, c, d)
```

This comma separates list elements:

```
[b, c, d]
```

This comma is a functor:

`(b, c, d)` means



Coercion

If objects of two numeric types appear as operands, we "upgrade" the "lower" type.

Hierarchy in Fortran:

`integer` < `real` < `double` < `complex`

In Pascal:

`integer` < `real`

Logical expressions

Six comparison operators

equal	=	==	.EQ.
not equal	<>	/=	!= .NE.
less	<		.LT.
less or equal	<=	=<	.LE.
greater	>		.GT.
greater or equal	>=	=>	.GE.

Equality is well-defined for all types, but no natural ordering usually exists for non-scalar types.

```
function "<"(L, R: date) return boolean is
earlier: boolean := true;
begin
   if L.year > R.year then
      earlier := false;
   elsif L.year = R.year then
      if L.month > R.month then
         earlier := false;
      elsif L.month = R.month then
         if L.day >= R.day then
            earlier := false;
         end if;
      end if;
   endif;
   return earlier;
end;
```


Testing that involves sets

equality: =, <>
membership: `in`, `not in`
inclusion (only in Pascal): <=, >=

Logical operators

Pascal: `not`, `and`, `or`

Ada: `not`, `and`, `and then`, `or`, `or else`, `xor`

Short-circuit operations are based on these facts:
`true or else` **ANYTHING** ≡ `true`
`false and then` **ANYTHING** ≡ `false`

Evaluating `P and Q and R` may mean evaluating all of `P`, `Q`, `R`, or stopping after the first `false`.

`P and then Q and then R` must stop after computing the first `false`.

Similarly, `P or else Q or else R` must stop after computing the first `true`.

```
A: array(1..10)of integer;
```
Unsafe: `if n>10 or (A(n)=0) then --`
Safe: `if n>10 or else (A(n)=0) then --`

Assignment

A single assignment is obvious:

```
target := expression
target = expression
target ← expression
```

Multiple assignment is more interesting.
PL/I: `A, B := EXPR;`
Algol 60: `A := B := EXPR;`

(1) Find the value of EXPR.
(2') Assign this value to A, then B.
(2") Assign this value to B, then A.

This is not quite unimportant. Consider

```
I := 5;   A[I] := I := 10;
```

The order in which target addresses are found also matters!

(1) Find all target addresses.
(2) Find the value of EXPR.
(3) Assign this value to A and B.

With this method, `A[5] := 10`.

(1) Find the value of EXPR.
(2) Find target addresses left-to-right, assign the value to every address.

With this method, still `A[5] := 10`.

(1) Find the value of EXPR.
(2) Find target addresses right-to-left, assign the value to every address.

With this method, `A[10] := 10`.

This statement in C is not a multiple assignment:

```
A = B = EXPR;
```

Here, `B = EXPR` has a value (the value of EXPR) that is next assigned to A: the assignment operator in C associates right-to-left.

Another syntactic addition in C: mixing assignment with arithmetic.

```
A += B;  means  A = A + B;
A *= B;  means  A = A * B;  etc.
```

Finally, we can have conditional targets (in C++):

```
(x != 0 ? y : z) = 17;
```

or (even less readable):

```
x ? y : z = 17;
```

Summary

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....