Operators and precedence

Contents

General concepts

Precedence

It is customary in mathematics to omit parentheses to simplify complex expressions that use binary operators. For example we often write "$3 + 2 + 1$" instead of the more complex "$(3 + 2) + 1$".

However as soon as we use different types of operators, the question immediately arises: is "$1 + 2 \times 3$" equal to "$(1 + 2) \times 3$" or "$1 + (2 \times 3)$"? In mathematics at least, we use *precedence rules* to disambiguate: we say that "multiplication has a *higher precedence* than addition", so the sign $\times$ binds to its operands "tighter" than the sign $+$.

Since we have multiple operators in arithmetic at least, we use a *precedence table* to determine which operators bind tighter than others:

Arithmetic operator	Precedence (higher binds tighter)
$\times \div$	4
$+ -$	3
$= < > \leq \geq$	2
$\Leftrightarrow$	1

Thanks to this table, we know that "$1 < 2 + 3$" is a short way to write "$1 < (2 + 3)$" and not "$(1 < 2) + 3$", because $+$ has a higher precedence than $<$.

Grouping order, fixity in math

When we use the *same* operator two or more times, another question arises: should we group "$1 + 2 + 3$" as "$(1 + 2) + 3$" or as "$1 + (2 + 3)$"? For the $+$ operator, this does not matter much, because addition is associative: we get the same value in either case. However, the situation is different with $-$: we get different values for "$1 - 2 - 3$" if we group like this: "$(1 - 2) - 3$" (equals -4) or like this: "$1 - (2 - 3)$" (equals 2). In math there are a couple of well known operators that are not associative: $-$, $\div$ and exponentiation ($(x^y)^z \neq x^{(y^z)}$).

For this situation, in mathematics we nearly always choose for *left fixity*: when we use the same operator two or more times in a row, we start grouping from the left. This word "fixity" means "grouping order".

Predecence and associativity in programming

Like in math, nearly all programming languages have operator precedence rules that make it possible to omit parentheses in many cases. Most often, all operators in a programming language that represent a mathematical operation have a similar precedence as in math. However, there are also other programming operators than in math, so for every programming language you will need to have a look at its *operator precedence table*.

For example, a part of the Java operator precedence table looks like this:

Operator	Precedence
`* / %`	3
`+ -`	4
`< >`	6
`==`	7

Note the precedence level is sometimes represented, like in the table above, by a number where a lower number means a higher precedence. For example priority "1" may mean the operator binds tighter than all other operators with a larger priority number. As a rule of thumb, read the description of the table! Also, usually the highest precedence operators are listed first.

In addition to precedence levels, programming languages also define a fixity for most operators. This is called *associativity* in the context of programming. For example we say that the operators `+` and `*` in Java are "left-associative" to mean that they are grouped from the left.

The following links provide operator precedence tables for different languages. **Check them out and see where they overlap and where they differ.**

- Operator precedence in Java
- Operator precedence in C++
- Operator precedence in Python
- Operator precedence in R
- Operator precedence in Matlab and Simulink

Note in particular that some language place ordering comparison (`< >`) at the same level as the equality comparison, and others not.

Right associativity

The reason why we need to care a bit more about associativity in programming than in math is that *some operators are right-associative*.

For example, in Python, the arithmetic exponentiation noted `**` is right-associative: the expression "`2**3**4`" is really equivalent to "`2**(3**4)`". (Java does not have an operator for this.)

In Java (and other languages from its family including C and C++), there are two imporant constructs that are right-associative: assignment and `if-else`.

Right associativity of assignments

The assignment in C, C++, Java and other languages is defined to be right-associative in order to simplify the assignment of multiple variables to the same value. So instead of writing:

```
x = 3;
y = 3;
z = 3;
```

we can write:

```
x = y = z = 3;
```

Which is equivalent to "`x = (y = (z = 3))`". This works because the "result" of an assignment is the variable on the left of the equal sign. So "`z = 3`" sets `z` to the value 3 then gives `z` back into "`y = z`", which changes `y` then gives `y` back to "`x = y`". Of course all this would not work if the assignment was left-associative.

Right associativity of `if-else`

This is a dangerous pitfall which need particular attention. Consider the following program fragment:

```
int x = 1;
if (1 > 6)
   if (3 > 2)
     x = 2;
else
   x = 3;
```

What is the resulting value for `x`? The proper answer is 1, not 3! This is because `if-else` is right-associative: the `else` part must be grouped with the first "`if`" found from the right. So the example above is really equivalent to:

```
int x = 1;
if (1 > 6)
{
   if (3 > 2)
     x = 2;
   else
     x = 3;
}
```

Combined operators

Most programming language provide *combined operators* to shorten expressions.

The most well-known are "`<=`" and "`>=`", from mathematics. We can draw the following equivalence table:

Operation	Equivalent to
`a <= b`	`(a < b) || (a == b)`
`a >= b`	`(a > b) || (a == b)`
`a != b`	`!(a == b)`

Combined assignment and arithmetic

In nearly all programming languages with an assignment operator, the following pattern happens often in code:

<variable> = <variable> <operator> <value>

Where the same variable name is expressed on the left and right side of "`=`". For example:

```
i = i + 1
a = a * 2
z = z - 3
```

Since this pattern is so common, most language will provide combined assignment operators of the form:

<variable> <operator>= <value>

For example in Java, C and C++ the following equivalences hold:

Operation	Equivalent to
`a += b`	`a = a + b`
`a -= b`	`a = a - b`
`a /= b`	`a = a / b`
`a *= b`	`a = a * b`
`a %= b`	`a = a % b`

Pre- and Post- increment and decrement

The C language, which Java inherits from, also has two additional operators: `++` and `--`. They each exist in two forms: a *prefix* form "`++x`" and a *suffix* form "`x++`", for a total of 4 operators. They are defined as follows:

Pre-increment and pre-decrement:

Syntax:

`++` <variable>
`--` <variable>

Semantics:

Equivalent respectively to "<variable> `+= 1`" and "<variable> `-= 1`": the variable is incremented (resp. decremented) by 1, then the variable is returned as result into the enclosing expression.

For example:

```
int x = 42;
int y = ++x;
System.out.printf("%d %d\n", x, y); // prints 43 43
```

The prefix form is the simplest to understand at first contact, however in practice the suffix form is more commonly used:

Post-increment and post-decrement:

Syntax:

<variable> `++`
<variable> `--`

Semantics:

1. the current value of the variable is saved.
2. the variable is incremented (resp. decremented) by 1.
3. the value saved in step #1 is returned as result into the enclosing expression.

For example:

```
int x = 42;
int y = x++;
System.out.printf("%d %d\n", x, y); // prints 43 42
```

The name "pre-increment" comes from Latin "pre" for "before": the variable is incremented before its value is given back in the enclosing expression. In "post-increment" the variable is incremented "after" its value is given back in the enclosing expression.

Important concepts

- *operator precedence*;
- *fixity* in math and *associativity* in languages;
- how to read a precedence table;
- right associativity of assignments in Java;
- right associativity of `if-else` in Java;
- combined operators;
- pre- and post- increment and decrement.

Further reading

- Think Java, section 2.7 (pp. 19-20), section 8.8 (p. 97)
- Introduction to Programming, section 2.5 (pp. 47-51)
- Absolute Java, Appendix 2 (p. 1141), increment & decrement operators (pp. 30-33)

Copyright and licensing

Copyright © 2014, Raphael 'kena' Poss. Permission is granted to distribute, reuse and modify this document according to the terms of the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/.