Operator Precedence Parsing

Table of Contents

What is parsing?

Parsing means processing structured data to identify its different parts.

For example, when a web browser gets a web page from a server, it first parses the html. It will probably find the title element, which will be text like <title>My Web Page</title>. It will also probably find a link to a style sheet, like <link rel="stylesheet" href="myStyles.css">. It will then send another GET request to the server, to fetch the style sheet. After parsing the html, it can go forward to displaying it.

Program statement parsing

In this context the parsing applies to an *expression* in a programming language statement, like “2+3*y”

An interpreter will *evaluate* this expression. In other words, calculate its value. A compiler will generate code which, when the object code is executed, will evaluate it.

But first the expression must be parsed.

An *operator precedence parser* OPP is one way to do this.

Operators and operands

An *operator* is for example + or – or * or /. An operator carries out some operation, like addition.

The values that the operation is applied to are the *operands*. For example in 2+3, the operands are 2 and 3.

We might also have expressions like 2*x+y. There are three operands here : 2, x and y. The x and y are *variables*.

We might also have operators like >, with expressions like x>y. These are *relational operators*, and the result is true or false. This is *boolean type*. In this introduction we will not cover the idea of type, and have everything as number type (and ignore integer and floating point values).

Precedence

In normal maths, we do some operators before others. For example 2+3*4 is 14, not 20, because we do the * before the +. We say * has higher precedence than +. We can use brackets to over-ride this, so (2+3)*4 is 20.

The problem

How do we write program code to do this? Suppose the expression is

2+3+4

We can simply go through this left to right, and when we meet an operand, do it. So we get 2+3, = 5, then +4, so 9.

The problem is, for example,

2+3*4

Here we have to do the 3*4 first, and go back to do the + 2. So we cannot just work left to right.

We also have

(2+3)*4

where we should do the add first.

This is the problem.

Tokens

Suppose our expression is

1.2+7.9

This is seven characters. But its only three ‘things’ - two numbers and an add.

The point is that sometimes, one ‘thing’ is several characters. So we do not want to process the expression one character at a time. We want to handle it one thing at a time.

Another example – an expression like

tax+1.5

Again this is just three things, namely one operator and two operands. One operand is a variable, with name ‘tax’. It is often the case that variable names are several characters long.

The things are properly called *tokens*, and the first stage in this process is tokenisation – replacing characters by tokens.

So it changes the character string

1.2+7.9

into 3 tokens

<number 1.2><operator +><number 7.9>

Associativity

Suppose we have

9+5+2

Should we do the left addition first, or the right?

(9+5)+2 =14+2 = 16

9+(5+2) = 9 +7 = 16

It makes no difference. (a+b)+c = a+(b+c). We say that addition is *associative*.

Suppose we try it with subtraction. For example 9-5-2

should we work out (9-5)-2, or 9-(5-2)?

(9-5)-2 = 4 - 2 =2

9-(5-2) = 9 - 3 =6

So subtraction is not associative. (a-b)-c is not equal to a-(b-c)

Multiplication is associative. For example 2*(3*4) = (2*3)*4

Division is not associative. 2/(3/4) = 8/3, but (2/3)/4 = 2/12 =1/6

If an operation is not associative, it is a matter of convention which is done first. In other words there is no 'real' answer which we do first. It is just the usual version. For subtraction and division, this is left-most first. So 9-5-2 is (9-5)-2 = 4-2 = 2. We say they are left-to-right associative, or *left associative*.

In real programming x=a-b-c might be seen as bad style, since it is not very clear which is to be done first. x=(a-b)-c is clearer.

= as an operator

We might think of a statement as like

x = <expression>

But we can also think of = as an operator. Then

x = y

has a value y. But it has a *side-effect*, that it makes variable x get the value y.

So

x=5*3

has the value 15, and the side-effect of making x to be 15. In languages where this applies (in C for example) we can then say

y = x = 5*3;

This makes x to be 15, and has value 15, so it is the same as

y = 15

= is *right-associative*. So if we say

y = x = b

we mean to do 'x=b' first, and the 'y=' second.

The symbol table

If we have variables, the system must - 'remember' their values, stored in memory, and this is done using a symbol table. This is a set of name-value pairs. So

x=5

looks for x in the symbol table and makes its matching value to be 5. Then

y = x

looks in the symbol table for x, and replaces it by the value found, so 5 or whatever.

Some languages (like C) require variables to be declared before being used. That puts them in the symbol table, starting either with default values or undefined.

Languages that do not (such as Python) will add a variable to the symbol table, if it occurs on the right hand side RHS. So

x = 5

as the first reference to x will add it to the symbol table, but

y =x

will produce the error that x is undefined.

Exactly how the symbol is stored depends on the language, and the implementation (in other words, how the interpreter or compiler and runtime is written). For local variables, for example, this is somehow on the stack (as opposed to the heap for global variables).

OPP basic algorithm

The first stage is tokenisation, to convert a string of characters to a list of tokens.

We then process those tokens, left to right, using (in this version) two stacks, one for values and one for operators.

The process involves a *reduce* operation. This

1. pops an operator off the operator stack
2. pops two values off the value stack (if its a binary operator)
3. applies the operator to the values
4. pushes the result back onto the value stack.

So for example, if the stacks, with the top on the right, are

Operator stack: <other data> +

Value stack: <other data> 2 3

We pop + off the first stack, 2 and 3 off the second, work out 2+3, and push 5 back on the value stack. We get

Operator stack: <other data>

Value stack: <other data> 5

As an introduction, ignoring several issues, we deal with

2 + 3 + 4

Values (2 3 and 4) are pushed on the value stack when reached.

When we get an operator (+):

if the operator stack is empty, we push it on the operator stack

else we do a reduce

At the end of the input, we do a reduce. The expression value is the single value left on the value stack. So:

Input	Value stack	Operator stack	Action
2 + 3 + 4	2		Push value
2 + 3 + 4	2	+	Push operator onto empty stack
2 + 3 + 4	2 3	+	Push value
2 + 3 + 4	5		Reduce
2 + 3 + 4	5	+	Push operator onto empty stack
2 + 3 + 4	5 4	+	Push value
	9		End of input - reduce

Precedence

One improvement on this is how to deal with precedence. For example, with

2+3*4

we must do the * first, and + later, but with

2*3+4

we must do the * first.

We do this as follows. When we reach an operator token, and there is already an operator on the stack, we compare their precedence.

If the new operator has higher precedence, we push it onto the stack.

If it is lower precedence, we reduce (so the operator on the stack will be used up), then push the new one.

For example with 2+3*4:

Input	Value stack	Operator stack	Action
2 + 3 *4	2		Push value
2 + 3*4	2	+	Push operator onto empty stack
2 + 3 *4	2 3	+	Push value
2 + 3 * 4	2 3	+ *	Push operator (higher precedence)
2 + 3 * 4	2 3 4	+ *	Push value
2 + 3 * 4	2 12	+	Reduce (do * first)
	14		Reduce (then do +)

We get + * on the stack. We take things off the top of the stack to do them, so we will do the * before the +.

With 2*3+4

Input	Value stack	Operator stack	Action
2 * 3 + 4	2		Push value
2 * 3 + 4	2	*	Push operator onto empty stack
2 * 3 + 4	2 3	*	Push value
2 * 3 + 4	6		Reduce (lower precedence)
2 * 3 + 4	6	+	Push operator onto empty stack
2 * 3 + 4	6 4	+	Push value
	10		End of input - reduce

When we reached the +, it had lower precedence, so we did the * first, and + later.

Associativity

If the current token is the same operator as on the top of the stack, we must look at associativity. For example with

y = x = 3+2

we must do the right-hand = first. This will work out x=3+2 and get 5. This has the side effect of putting the value of x to be 5 in the *symbol table*; but primarily it has a value 5, so the left-hand = in effect is

y = 5

If we did the left = first, we would need to evaluate

y = x

and if x does not have a value at that stage, that is meaningless.

Brackets

For example we need

2*(3+4)

to be 14, doing the + before the *.

We do this as follows:

- When we reach a (we push it onto the operator stack
- When we reach the matching), we reduce until the top of operator stack is (. Then we delete the (and).

For example:

Input	Value stack	Operator stack	Action
2*(3+4)	2		Push value
2*(3+4)	2	*	Push operator onto empty stack
2*(3+4)	2	* (	Push (
2*(3+4)	2 3	* (	Push value
2*(3+4)	2 3	* (+	Push operator - (has lower precedence
2*(3+4)	2 3 4	* (+	Push value
2*(3+4)	2 3 4	* (+	Loop with) - reduce
2*(3+4)	2 7	* (	Have (and) - drop both
2*(3+4)	2 7	*	Reduce
2*(3+4)	14		Accept

This will also work with nested brackets. For example

2*((3+4)*5)

The green (will get pushed on the stack on top of the red one. The green) will empty the stack back to the green (, and both will go. Then we go forward to the red), and this will work back to the red (.