# Operator Precedence Parsing

## What we are trying to do

If we are writing a compiler or interpreter, what code should we produce for statements like

x = 1 + 2 + 3

The obvious way to do it is to go through the expression 1 + 2 + 3 left to right, and carry out the operations as we come to them. So first we add 1 and 2 to get 3, then add the 3 to get 6.

## The problem

But suppose we need to do

x = 1 + 2 * 3

To work this out correctly, we must first do the 2 times 3, then the add. As we go through it left to right, when we reach the add how do we know not to do it yet, since we have not yet reached the * ?

Another example

x = (1+2)*(3+4)

Here we should first add 1 and 2, then 3 and 4, then do the multiply. In other words the order of execution moves back and forwards through the expression.

This is the problem.

## Concepts involved

## Tokens and symbols

In

x = 1 + 2 * 3

each 'thing' in the input is one symbol. But we might have

price = cost + 1.34 +profit

In the expression we have 5 things:

'cost' '+' '1.34' '+' 'profit'

Some of those things are single symbols ( like + ) but others are several symbols (like 1.34 is 4 symbols).

The 'things' are usually called *tokens*. The first stage in the solution is to convert the input stream (like “cost + 1.34 +profit” ) into a list of tokens (here 5 tokens, cost + 1.34 + profit ). This is done by software called a *tokeniser*.

To focus on the parser, we will just use examples where 1 token is 1 symbol, like (1+2)*(3+4)

## Operators and operands

*Operators* are tokens like + - * and /. *Operands* are like 1 and 2 and 3 and 4. An operator does something with operands - so 1 + 2  is the operator + working on the operands 1 and 2.

In real code we might often have:

x = a * b

In other words the operands might be *variables* not constants. So we have the additional step of somehow replacing the variable with the current value of the variable. To simplify things, we'll just use constants.

## Precedence

In

1+2*3

we must do the * before the +. We handle this by giving each operator a precedence level, and carry out operators in order of precedence.

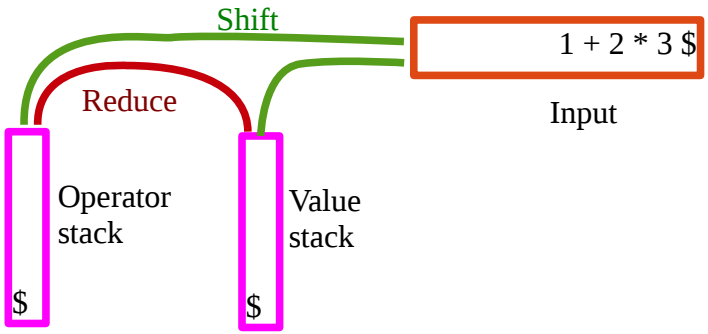
| Operators | Precedence (high predence is smaller number) |
|---|---|
| | 1 |
| F ( function - see below) | 2 |
| * / % | 3 |
| + - | 4 |
| ( ) | 5 |
| , | 6 |
| $ | 7 (end marker - see below) |

## Associativity

This is about what to do when we have 2 operators of equal precedence.

For example, what is
6 - 3 - 2
We could do the left - first. Then we get (6-3)-2 = 3 -2 = 1
Or we could do the right - first. Then we get 6 - (3-2) = 6-1 = 5
Different results, so we need some basis of knowing which to do.
An operator (call it ~ ) is *left-associative* if we do the left first, so
a ~ b ~ c = (a~b)~c
It is *right-associative* if we do the right one first, so
a  ~ b ~ c = a ~ (b ~ c)
Sometimes it makes no difference, as for + or *. But others, like -, it does.

# The parsing machine



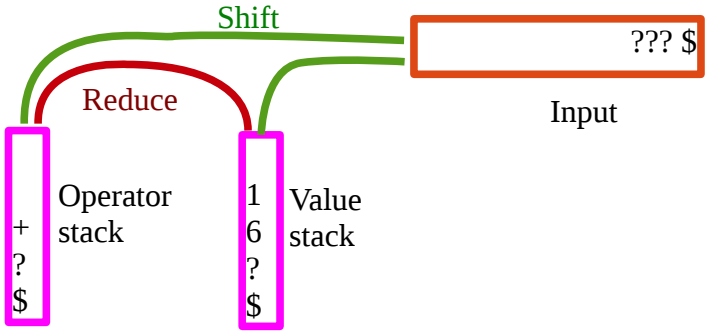

In effect an operator precedence parser is a machine, as shown here.

We have the input expression, and two *stacks*, one for values and one for operators. For convenience we start both stacks containing the terminal symbol, $, and put this at the end of the input expression.

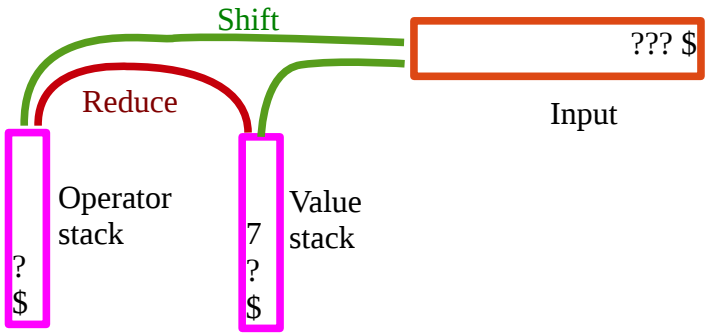
The machine can do two things - shift and reduce.

*Shift* means to move a token on to the appropriate stack

*Reduce* means to carry out the operator on the top of the operator stack, on values from the value stack, and put the result back on the value stack. For example if the machine is like this:




and we do a reduce, we pop the + off the stack, pop 1 and 6 off, add them, and put the result back, so we get:

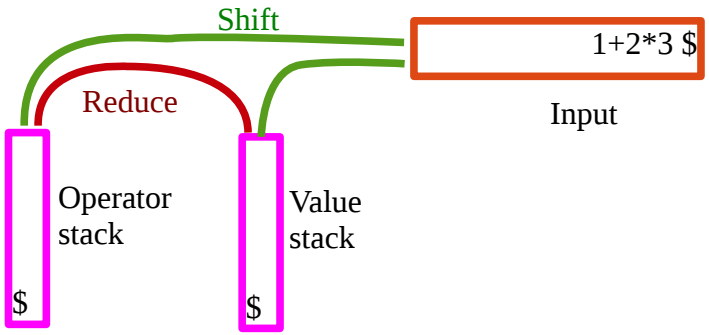



We go through the input left to right. All the operators must be executed - its a question of when. If they should be done later, we store them on the stack:

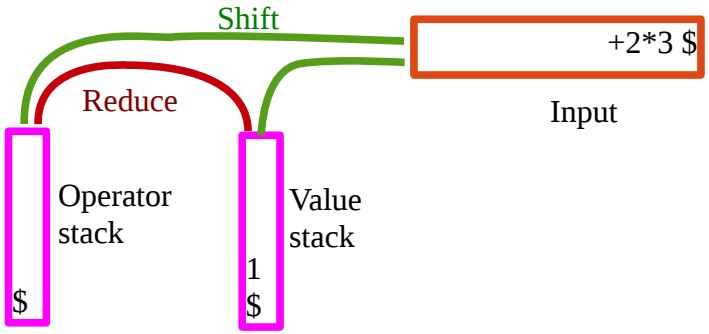
If next token is a value, Shift to the Value value stack
Else if next token is an operator, compare it with the top of the operator stack, and take appropriate *action*

We decide the *action* as follows

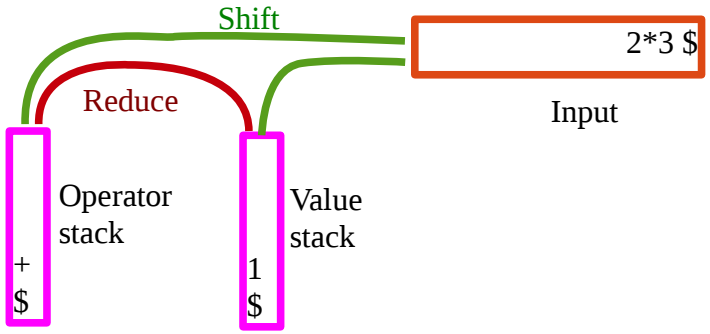
If the operators are different, then:
if the input has higher precedence, shift it to the operator stack - else reduce
else if they are the same
shift to operator stack for right associativity, reduce for left associativity
If the input and top of stack are both $, input has ended. A single value left on the stack is the expression value.
We trace this through with 1+2*3: Initially:



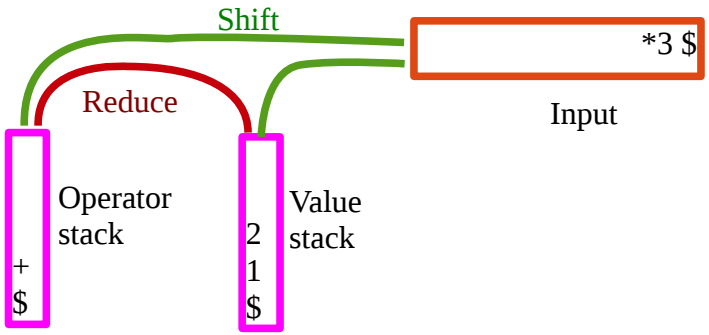

Input token is 1, a value, so Shift to the value stack:



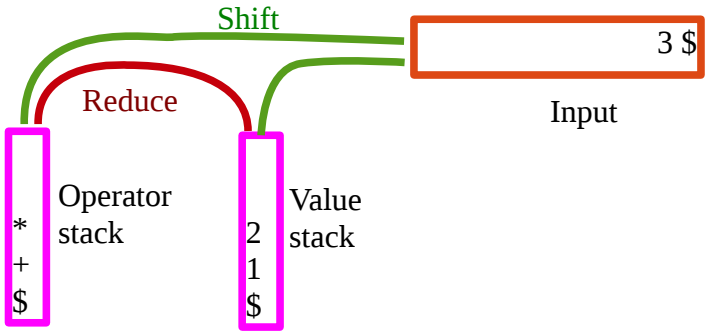

Input is +. Top of operator stack is $. + has higher precedence, so Shift:



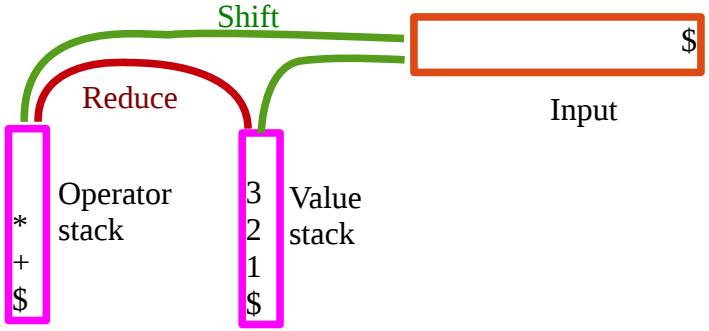

Input is 2. Shift to value stack



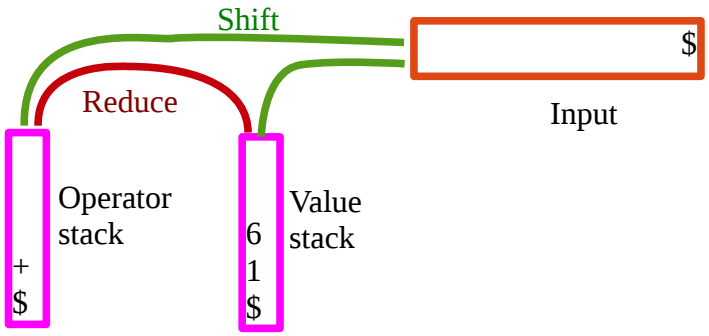

Input is *. Top of operator stack is +, which is lower precedence, so shift



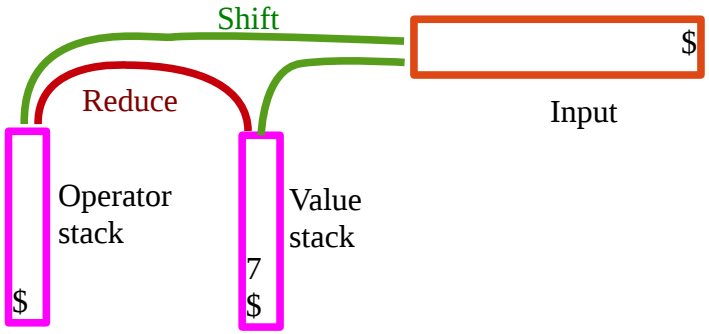

Shift the 3




Input is $, top of stack is *, which is higher precedence, so reduce:




Now $ and +, so reduce again:



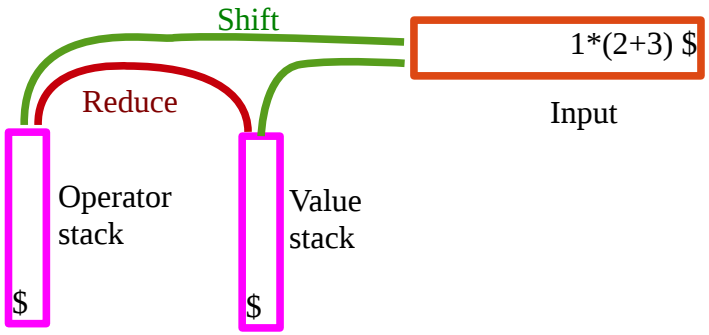

Now input is $, top of stack is $, so the action is 'Accept'. The machine stops, with one value on the value stack, the value of 1+2*3.

Why does it work? The crucial point is inputting the * with + on top of stack. The * has higher precedence, so we Shift. That puts the * above the +, so when we reduce, the * is applied before the +, as required.

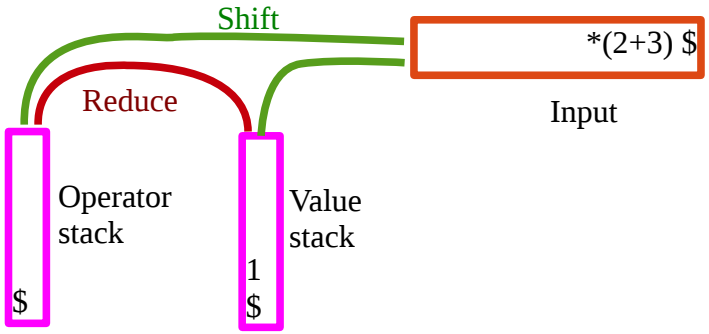
## Effect of brackets

We have a special rule for brackets. If the input is (, shift.  If the input is ), we reduce, until the top of stack is (. Then we discard the top of stack and the input )

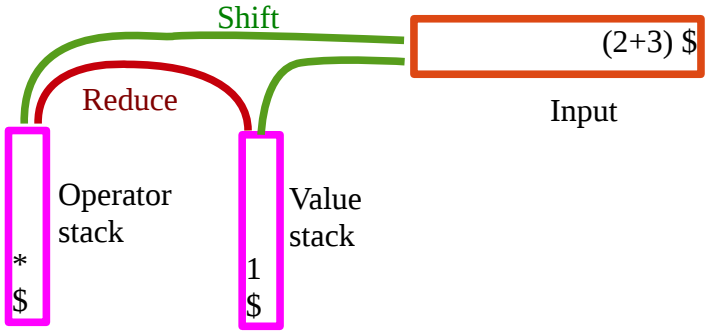
Suppose we trace the example 1*(2+3)



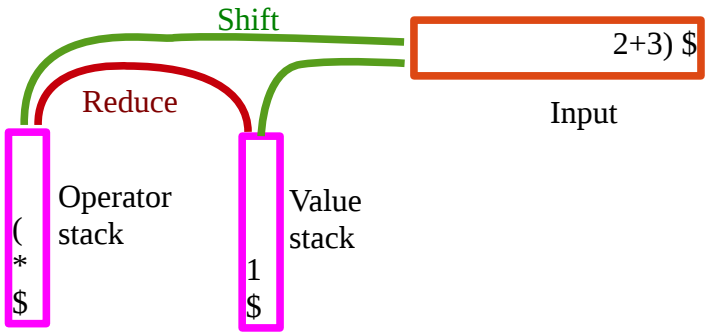

Shift 1



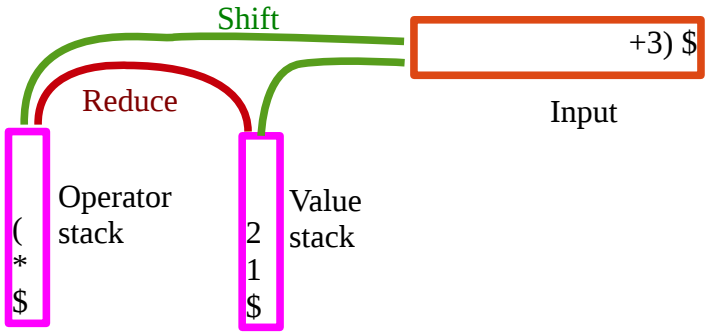

Shift *



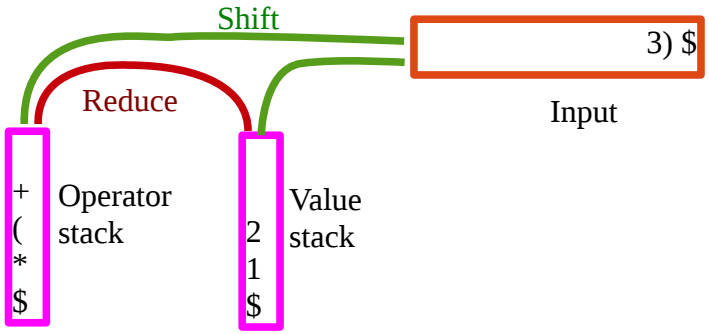
Shift (



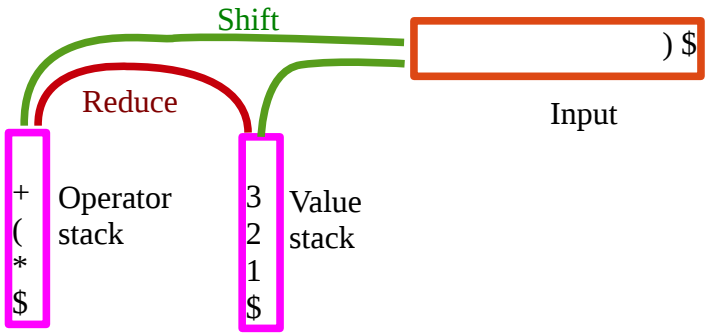

and 2



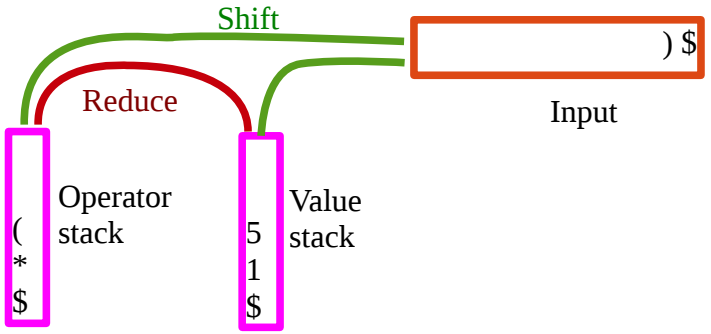

Shift +



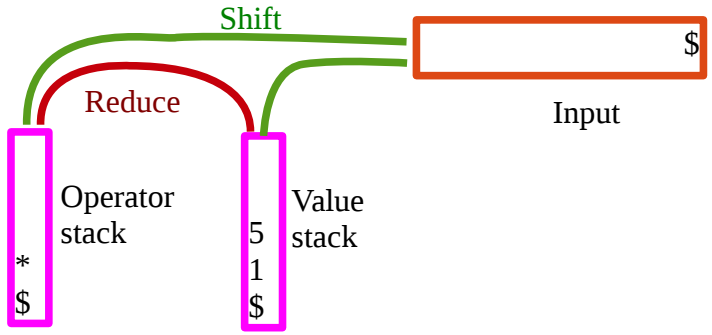

Shift 3



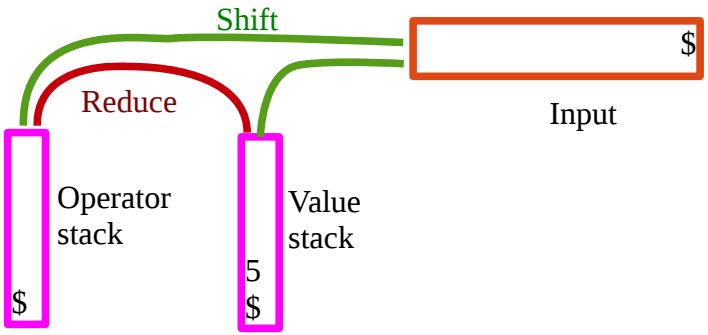

Now the special rule. Reduce repeatedly till top of stack is ( - here just once:




Then remove the ( and )



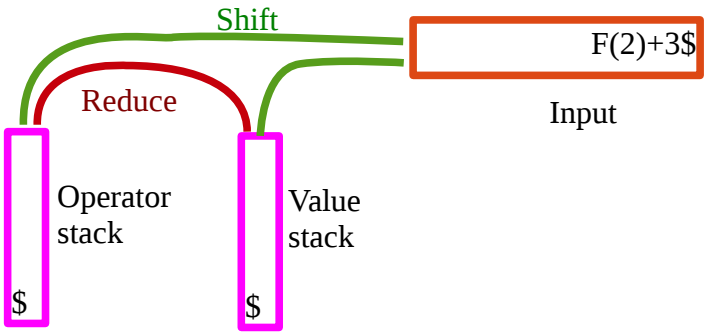

Reduce



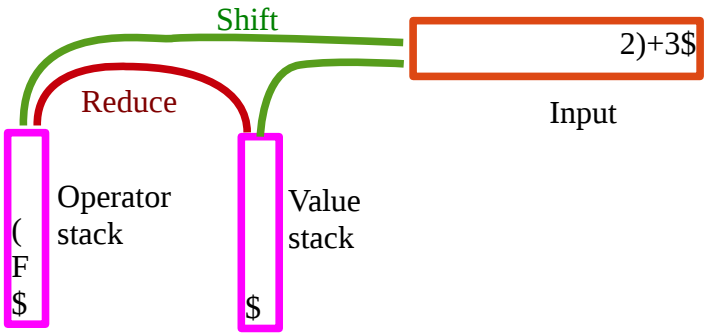

the Accept, with result 5.

## Handling functions

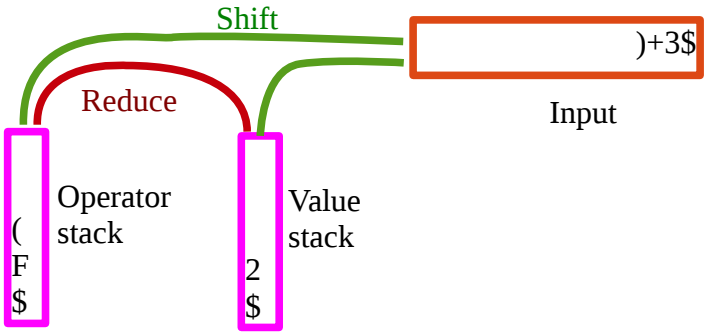
To start with, suppose we have just one function, named F, such that F(x)=2x, so F(2)+3 would be 7. Suppose we try F(2)+3



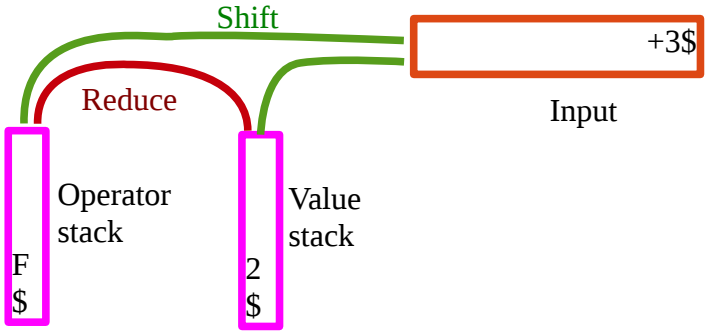
Shift  F and (



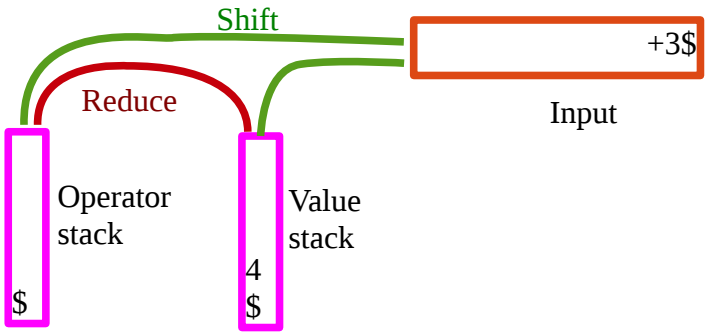

Shift 2



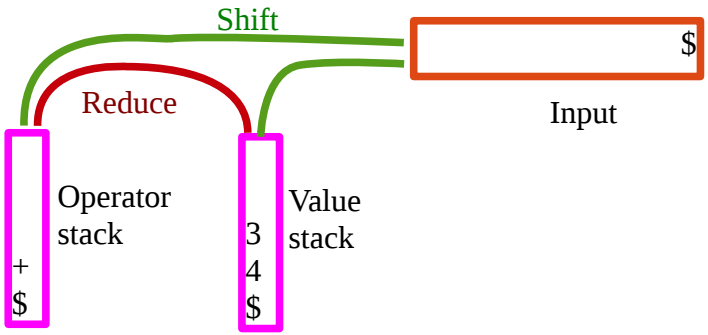

Nothing to reduce - remove the ( and )



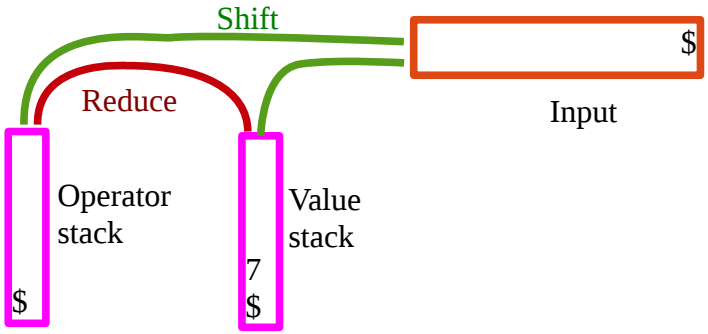
Function has higher precedence so reduce (apply the function on the argument on top of stack)




Shift the + then the 3




reduce and we've done.

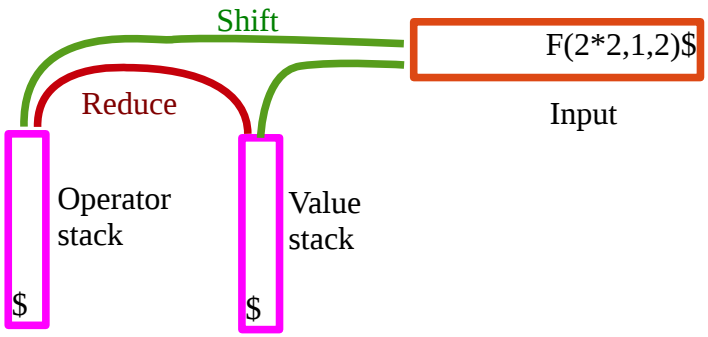


# Functions with several arguments

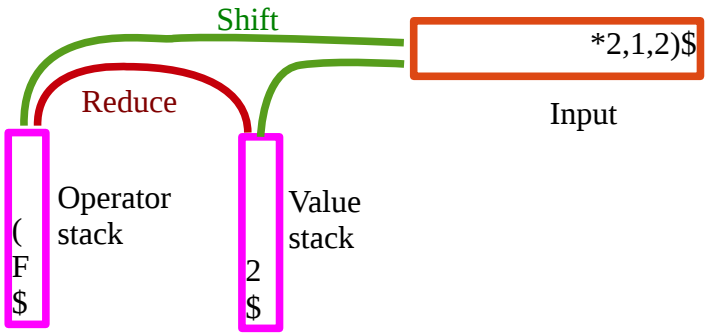
Instead suppose F takes three arguments, and F(a,b,c) = a-b-c, so F(2*2,1,2) is 1.

The rule for a comma on input is:
Reduce until ( is top of operator stack
Discard the ,

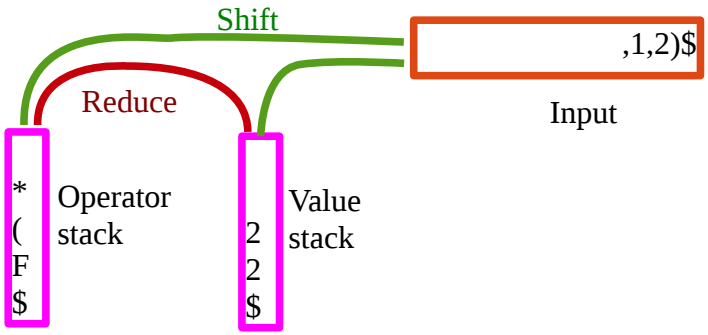
The start is



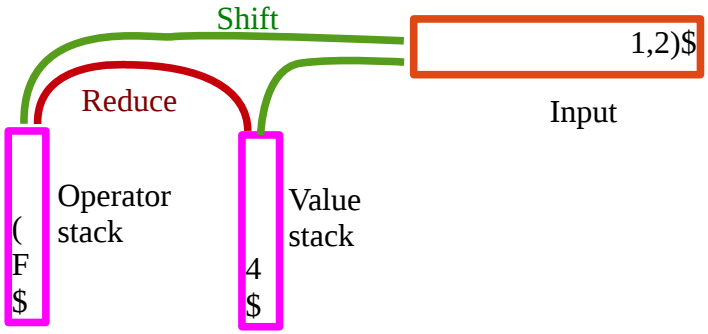

Shift the F, ( and 2



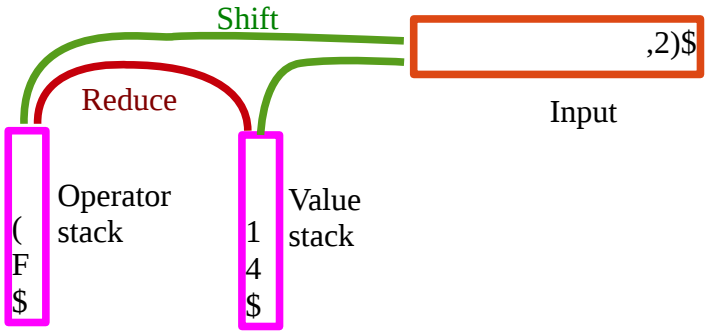

then * and 2



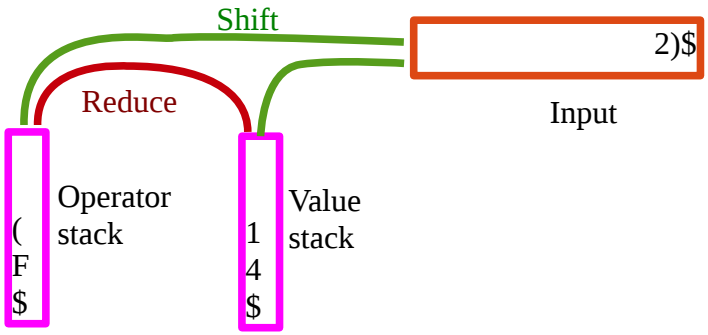

Apply the comma rule



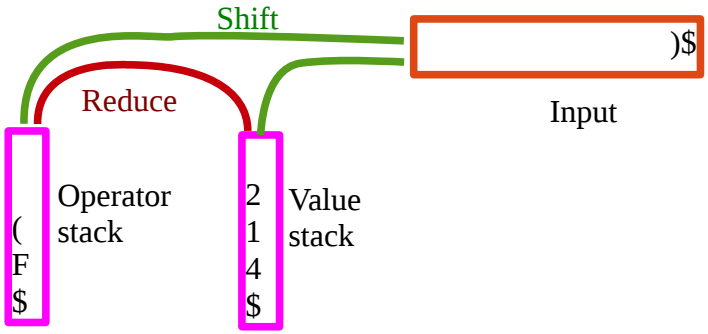

Shift 1



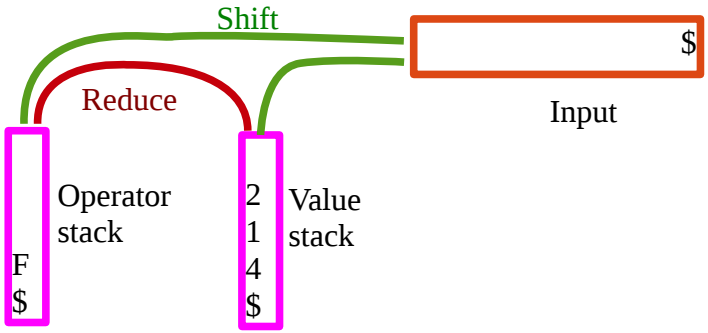

Comma rule (no reduction to do)



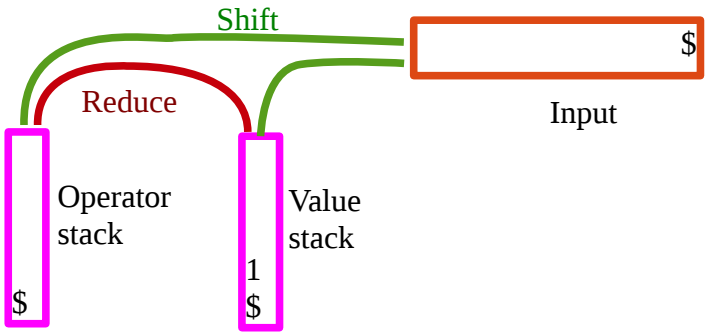

Shift 4




Brackets rule:




Reduce. F takes 3 arguments, so 3 values are popped off the stack (in reverse order) and processed




Why does this work? To evaluate F(a,b,c), we need to get F on the top of the operator stack, and a b and c on the value stack. Then on the reduce, F will pop off a b and c, apply F to them, and push the result back. But we might have something like F(a,b+c,d) - in other words the arguments may be expressions which need to be evaluated first. The rule for comma ensures these are evaluated first, by reducing anything on the operator stack down to the (.

In practice we obviously need more than one function. One way to do that, during tokenisation, is to have a single symbol, say F, for all functions, and have another field in the token as an integer for which function it is. When a function token is shifted, we shift the integer on the value stack. When we come to reduce the function, we must pop that value as well as the arguments, so we know which function it is.