Priority Techniques

How can we use stacks to compute arithmetic expression: reminder. To remind you how this is done, let us recall that there are three possible way to write an arithmetic expression line $2 + 3$:

- the usual way $2 \ + \ 3$ is called *infix* notation, since here the operation symbol $+$ is placed in between the two numbers;
- it is also possible to write $+\ 2\ 3$; this corresponds to the way we say it in English (and in many other languages): "add two and three"; here the operation symbol is placed before the numbers, so it is called *prefix* notation; this is, by the way, how mathematical functions are denoted: e.g., we write $\ln x$, the operation symbol before the number to which it is applied;
- finally, it is possible to write $2\ 3\ +$, with the operation symbol after the numbers; this is known as the *postfix* notation; it corresponds to some languages like German or Japanese in which the verb is usually at the end of the phrase; in English that would sound like "two and three add".

Historical note. Postfix notations are, for historical reasons, also known as *inverse Polish notations*. The reason is that in the 1920s and 1930s, Poland was the center of research in mathematical logic and what would not be called foundations of computations.

Polish researchers tried to describe computations in a formal way, and they found it convenient to describe functions like $a \ + \ b$ the same way we describe all other functions – by placing the operation symbol first, as $+\ a\ b$. These notations became known as *Polish notations*.

The inverse – when we place the operation symbol at the end – naturally became known as the *Inverse Polish notations*.

Which of three notations is best for computing? The main purpose of a computer is to compute fast. Let us compare the three notations from this viewpoint, taking into account that the computer reads the symbols in the program one by one.

Let us start with the infix notation $a \ + \ b$.

- When the computer reads a, it knows what to do: it fetches the value a from the corresponding place in the memory to the central processing unit.

- Now the computer reads the symbol +. The computer knows that it has to add a and some other number, but that number is not available yet. So, before reading the next number, the computer just waits and does nothing. Not the best way to use the computer: if we want it to be fast, we need to make sure that it is never idle, that it does not have to wait.

What about the prefix notation $+\ a\ b$? Same thing: when the computer reads the first symbol +, it does not know what to do, so this time is wasted.

For postfix notation $a\ b\ +$, the situation is different:

- first, we read a and fetch the number a;
- then, we read b and fetch the number b;
- finally, we read the operation symbol + and we add the two numbers which have already been fetched.

Clearly, from the viewpoint of computation speed, the postfix notation is the best. So, not surprisingly, this notation is what is used in compilers.

Resulting procedure: main idea.

- First, we transform the original expression into the postfix form.
- Then, we compute the value of the resulting expression.

Why stacks. For both procedures, we use stacks. Why stacks? Because, as we have mentioned when we started describing pushdown automata, a stack is a structure which is already automatically implemented in a computer.

How stacks are used to transform the expression into a postfix form: simplest no-parentheses case. In this procedure, we start with an empty stack. As we read the symbols of the arithmetic expression one by one:

- we form a postfix expression, and
- we push operation symbols into the stack and pop them from the stack.

At the end, the stack is empty, and we get the desired postfix expression.

The procedure is as follows:

- if you see a number (or a variable name), you copy it into the postfix expression;
- if you see an operation symbol, then you push this symbol into the stack, and also if the symbol of top of the stack has higher (or same) priority as this symbol, you pop this symbol from the stack to the newly formed postfix expression before pushing the new symbol into the stack.

First example. Let us illustrate this procedure on the example of the expression

$$2 - 3 \cdot 4$$

We start with an empty stack.

• First, we read the first symbol 2:

$$\underline{2} - 3 \cdot 4$$

This symbol is a number, so we copy it into the line for the postfix expression, which now takes the form

$$2$$

The stack is still empty.

• Next, we read the symbol $-$:

$$2 \underline{-} 3 \cdot 4$$

This is an operation symbol, so we push it into the stack. At this moment, there was nothing in the stack, so we have nothing to pop. We did not change the postfix expression, which still has the form

$$2$$

The stack now has the form

• Next, we read the symbol 3:

$$2 - \underline{3} \cdot 4$$

This symbol is a number, so we copy it into the line for the postfix expression, which now takes the form

$$2\ 3$$

The stack remains the same.

• After that, we read the symbol $\cdot$:

$$2 - 3 \underline{\cdot} 4$$

This is an operation symbol, so we push it into the stack. At top of the stack, we had subtraction. Subtraction has lower priority than multiplication, so we do not pop anything from the stack. Thus, the stack takes the form

$$3$$

• Finally, we read the last symbol 4:

$$2 \ - \ 3 \cdot \underline{4}$$

This symbol is a number, so we copy it into the line for the postfix expression, which now takes the form

$$2\ 3\ 4$$

The stack remains the same as before.

• The line ended, so we read the end-of-line symbol – which in a computer indicated the end of line. This symbol has the lowest priority – in the sense that all operations described in the line have to be performed first. Thus, we pop the top symbol · from the stack to the postfix expression which now takes the form

$$2\ 3\ 4\ \cdot$$

The stack now has only one symbol left:

$$\boxed{-}$$

The end-of-line has lower priority than subtraction as well, so we pop − as well. Now, the postfix expression has the desired form

$$2\ 3\ 4\ \cdot\ -$$

and the stack is empty.

First example – resulting postfix expression: $2\ 3\ 4\ \cdot\ -$

First example: summary. Let us summarize, step by step, how the above algorithm worked.

- The top line in this description contains all the symbols from the original expressions.
- The next line contains the resulting postfix expression.
- Below this line, under each operation symbol from the original expression, we list the contents of the stack after we read this symbol, top symbol first.

In these terns, the above algorithm can be represented as follows:

2	−	3	·	4		
2		3		4	·	−
	−		·		−	
			−			

Second example. Let us now consider the expression

$$2 \cdot 3 - 4$$

We start with an empty stack.

• First, we read the first symbol 2:

$$\underline{2} - 3 \cdot 4$$

This symbol is a number, so we copy it into the line for the postfix expression, which now takes the form

$$2$$

The stack is still empty.

• Next, we read the symbol $\cdot$:

$$2 \underline{\cdot} 3 - 4$$

This is an operation symbol, so we push it into the stack. At this moment, there was nothing in the stack, so we have nothing to pop. We did not change the prefix expression, which still has the form

$$2$$

The stack now has the form

$$\boxed{\cdot}$$

• Next, we read the symbol 3:

$$2 \cdot \underline{3} - 4$$

This symbol is a number, so we copy it into the line for the postfix expression, which now takes the form

$$2\ 3$$

The stack remains the same.

• After that, we read the symbol $-$:

$$2 \cdot 3 \underline{-} 4$$

This is an operation symbol, so we push it into the stack. At top of the stack, we had multiplication. Multiplication has higher priority than subtraction, so pop $\cdot$ from the stack into the postfix expression before pushing $-$ there. Then, the postfix expression takes the form

$$2\ 3\ \cdot$$

The stack takes the form

$$\boxed{-}$$

• Finally, we read the last symbol 4:

$$2 \cdot 3 - \underline{4}$$

This symbol is a number, so we copy it into the line for the postfix expression, which now takes the form

$$2\ 3\ \cdot 4$$

The stack remains the same as before.

• The line ended, so we read the end-of-line symbol – which in a computer indicated the end of line. This symbol has the lowest priority – in the sense that all operations described in the line have to be performed first. Thus, we pop the top symbol − from the stack to the postfix expression which now takes the form

$$2\ 3\ \cdot\ 4\ -$$

The stack is now empty, so this is the resulting postfix expression.

Second example – resulting postfix expression: 2 3 · 4 −

Second example: summarizing.

2	·	3	−	4	
2		3	·	4	−
	·		−		

Practice. Try this algorithm on other arithmetic expressions.

What if we have parentheses? By definition, everything inside the parentheses has to be performed first. This determined the corresponding priorities. Let us explain this on the example of an expression

$$(\ 2\ -\ 3\ )\ \cdot\ 4$$

We start with an empty stack.

• First, we read the first symbol (:

$$\underline{(}\ 2\ -\ 3\ )\ \cdot\ 4$$

This symbol is not a number and not a variable, so it is an operation symbol, so we push it into the stack. The stack now has the form:

$$\boxed{(}$$

• Next, we read the symbol 2:

$$(\underline{2} \ - \ 3) \ \cdot \ 4$$

This symbol is a number, so we copy it into the line for the postfix expression, which now takes the form

$$2$$

The stack remains the same.

• Next, we read the symbol −:

$$(\ 2 \ \underline{-} \ 3) \ \cdot \ 4$$

This is an operation symbol, so we push it into the stack. Any operation inside the parentheses has to be performed before anything else. In priority terms, this means that any operation has higher priority than the opening parenthesis. Thus, we do not pop (from the stack. The postfix expression remains the same, and the stack takes the form

Note. To decide what has higher priority: (or −, we used common sense. Of course, computers do nor use common sense: they simply have a table (filled by the programmer who designed the compiler) that describes, for each pair of operations, which of them has higher priority.

Example (cont-d).

• Next, we read the symbol 3:

$$(\ 2 \ - \ \underline{3} \ ) \ \cdot \ 4$$

This symbol is a number, so we add it to the postfix expression, which now takes the form

$$2 \ 3$$

The stack remains the same.

• Next, we read the closing parenthesis):

$$(\ 2 \ - \ 3 \ \underline{)} \ \cdot \ 4$$

This is an operation symbol, so we push it into the stack. Every operation in parentheses has to be performed before everything else. In priority terms, this means that any operation has higher priority than). So, before pushing, we pop the top symbol − from the stack into the postfix expression, which takes the form

$$2 \ 3 \ -$$

Now, the stack has the following form:

This means, in effect, that we have an expression (), i.e., we have nothing to do. Thus, if we have these two symbols on top of the stack, they can both be canceled. When we cancel them, the stack will be empty.

• Next, we read the multiplication symbol:

$$(\ 2\ -\ 3\ )\ \underline{\cdot}\ 4$$

This is an operation symbol, so we push it into the stack. The stack was empty, so there is nothing to pop. Thus, the stack has the following form:

• Next, we read the last symbol 4:

$$(\ 2\ -\ 3\ )\ \cdot\ \underline{4}$$

This is a number, so we add it to the forming postfix expression:

$$2\ 3\ -\ 4$$

The stack remains the same.

• Finally, just like in the first example, we pop − from the stack, and get the final postfix expression:

$$2\ 3\ -\ 4\ \cdot$$

Resulting postfix expression: $2\ 3\ -\ 4\ \cdot$

How we got this postfix expression: summary.

(	2	−	3	)	·	4	
	2		3	−		4	·
(		−		)	·		
		(		(			

Practice. Apply this algorithm to other arithmetic expressions.

How to compute the value of a postfix expression: algorithm. We start with an empty stack and read the symbols one by one.

- if the symbol that we currently read is a number, we push it into the stack;
- if the symbol that we currently read is an operation symbols, then we:

- pop two top numbers from the stack,
- apply the corresponding operation to the second and the first of the popped numbers (note that the order is important), and
- push the result into the stack.

When we finish reading all the symbols, there should be one number in the stack. This number is the desired computation result.

First note. We have two algorithms:

- an algorithm for converting an arithmetic expression into a postfix form, and
- an algorithm for computing the value of the postfix expression.

In both algorithms, we use stacks, but we use them differently:

- in the first algorithm, stacks contain *operation symbols*, while
- in the second algorithm, stacks contain *numbers*.

Second note.

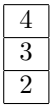
- Most operations are binary, so we pop two top numbers.
- For unary operations – like unary minus or ++ – we pop one top number.
- For ternary operations, we pop three top numbers, etc.

Example. Let us consider the example of our first postfix expression $2\ 3\ 4\ \cdot\ -$.

• First, we read the first symbol 2 of this expression $\underline{2}\ 3\ 4\ \cdot\ -$. This symbol is a number, so we push it into the stack. Now, the stack takes the form

• Next, we read the next symbol 3 of this expression $2\ \underline{3}\ 4\ \cdot\ -$. This symbol is also a number, so we also push it into the stack. Now, the stack takes the form

• After that, we read the symbol 4 of this expression $2\ 3\ \underline{4}\ \cdot\ -$. This symbol is still a number, so we push it into the stack. Now, the stack takes the form



• The next symbol that we read is a multiplication symbol 2 3 4 $\underline{\cdot}$ $-$. So:

- we pop the two top numbers from the stack: 4 and 3,
- we apply the multiplication operation to the numbers 3 and 4, i.e., compute $3 \cdot 4 = 12$, and
- we push 12 into the stack.

Now, the stack takes the form

12
2

• The last symbol that we read is the subtraction symbol 2 3 4 $\cdot$ $\underline{-}$. So:

- we pop the two top numbers from the stack: 12 and 2,
- we apply the subtraction operation to the numbers 2 and 12, i.e., compute $2 - 12 = -10$, and
- we push the resulting number -10 into the stack.

Now, the stack takes the form

-10

• We have read all the symbols, so the number -10 that we have in the stack is the result of our computation of the original expression $2 - 3 \cdot 4$, for which 2 3 4 $\cdot$ $-$ is the corresponding postfix form.

Important note. At the last step, we first popped 12, then 2. According to the algorithm, we apply the operation $-$ to the second popped value (i.e., 2) and the first popped value (i.e., 12), and compute $2 - 12$.

If we apply this operation to the numbers in the wrong order, i.e., if we instead compute $12 - 2$, we will get a wrong result.

Example: summary.

2	3	4	$\cdot$	$-$
2	3	4	12	-10
	2	3	2	
		2		

What if we have variables instead of numbers? In the actual programs, we rarely have arithmetic expressions with numbers, we usually have arithmetic expressions with variables. In this case, we can still transform the arithmetic expression into a postfix form by using the exact same algorithm. For example,

the expression $a + b \cdot c$ will be transformed into a postfix expression $a\ b\ c\ \cdot\ -$. However, after that, we cannot simply compute the results. What we can do is list what operations need to be performed at each stage. This description is usually described in terms of the so-called *quadruples*

$$\text{op}\ \ v_1\ \ v_2\ \ r,$$

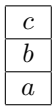
where:

- op is the symbol for the corresponding operation,
- v_1 and v_2 are the variables to which this operation should be applied, and
- r is where we place the result.

Intermediate results are usually stored in *registers* – this explains the letter r that we used. For simplicity, we will just use registers r_1, r_2, ... At the end, we will briefly explain that often, we do not need to use a new register every time: registers can be reused.

Let us illustrate this on the example of the postfix expression $a\ b\ c\ \cdot\ -$.

• The first three symbols in this expression – symbols a, b, and c – are variables that represent numbers. So, according to the above algorithm, we push them into the stack. As a result, the stack has the following form:



• The next symbol we read is the multiplication symbol $\cdot$ from the original expression $a\ b\ c\ \cdot\ -$. So, according to the general algorithm:

- we pop the top two numbers – in this case, the numbers c and b,
- we multiply the second popped number b to the first popped number c, computing $b \cdot c$; at the compile time, we do not know the values of b and c; so, instead of actually computing this value, we generate a quadruple that places this result into the first register r_1:

$$\cdot\ \ b\ \ c\ \ r_1$$

- we push the result – i.e., r_1 into the stack.

The stack now gets the form:

• The last symbol we read is the subtraction symbol $-$ from the original expression $a\ b\ c\ \cdot\ -$. So, according to the general algorithm, we:

- pop the top two numbers – in this case, the numbers r_1 and a,
- we perform the subtraction operation to the second popped number a and the first popped number r_1, computing $a - r_1$; at the compile time, we do not know the values of a and r_1; so, instead of actually computing this value, we generate a quadruple that places this result into the next register r_2:

$$- \quad a \quad r_1 \quad r_2$$

- we push the result – i.e., r_2 – into the stack.

The stack now gets the form:

$$\boxed{r_2}$$

The quadruples form exactly the sequence of operations that the computer needs to perform to compute the value of the desired expression $a + b \cdot c$:

$$\cdot \quad b \quad c \quad r_1$$

$$- \quad a \quad r_1 \quad r_2$$

So, we have indeed compiled the original expression.

Let us summarize:

a	b	c	$\cdot$	$-$
a	b	c	r_1	r_2
	a	b	a	
		a		

Note. In this case, once we get the value r_1 from the first register, we do not need this register anymore. So, instead of placing the result of subtraction into a new register r_2, we can place it into the same register r_1 – and thus, save the second register for other computations:

$$\cdot \quad b \quad c \quad r_1$$

$$- \quad a \quad r_1 \quad r_1$$

Practice. Try converting other arithmetic expressions into a sequence of quadruples.

What about a general program? What if instead of an arithmetic expression we have a general program? In this case, as we will show, the same method can be applied – and this is how most parts of the program are compiled.

Let us illustrate this on the example of the following simple program fragment:

$$x \ = \ a \ + \ b; \ y = Math.exp(x);$$

Let us first transform this expression into the postfix form.

As before, we will use common sense to decide which operation comes first, but a computer simply looks into the table.

• We start with the symbol x of the expression

$$\underline{x} \ = \ a \ + \ b; \ y = Math.exp(x);$$

Since this symbol described a number and not an operation, we add this symbol to the resulting postfix expression, which now takes the form

$$x$$

• Next, we see the assignment symbol

$$x \underline{=} a \ + \ b; \ y = Math.exp(x);$$

This is an operation symbol, so we push it into the stack. The stack was originally empty, so there is nothing to pop. Thus, the stack takes the form

$$\boxed{=}$$

• After that, we see the symbol

$$x \ = \ \underline{a} \ + \ b; \ y = Math.exp(x);$$

This symbol represents numbers, so we add it to the postfix expression, which now takes the form

$$x \ a$$

• The next symbol we read is the addition + form the expression

$$x \ = \ a \ \underline{+} \ b; \ y = Math.exp(x);$$

We push this symbol into the stack, but before that, we need to decide whether to pop = from the stack, i.e., whether the new operation + or the previous operation = has higher priority. This question is easy to answer: if we have a command

$$x = a + b;$$

we first add the numbers and only then perform the assignment. In other words, any operation after the assignment symbol has higher priority than the assignment. Thus, we do not pop anything, and the stack takes the form

$$\boxed{\begin{array}{|c|}\hline + \\ \hline = \\ \hline \end{array}}$$

• After that, we see the symbol

$$x \ = \ a \ + \ \underline{b};\ y = Math.exp(x);$$

The symbol b represents numbers, so we add it to the postfix expression, which now takes the form

$$x\ a\ b$$

• The next symbol in the expression

$$x \ = \ a \ + \ b\ \underline{;}\ y = Math.exp(x);$$

is the semicolon ; It is an operation symbol, so we push it into the stack. Everything before the semicolon has to be performed before we go to the next statement, so everything that was in the stack has higher priority that this symbol. Thus, before pushing, we pop all the symbols in the stack one by one and place them into the postfix expression. As a result, the postfix expression will take the form

$$x\ a\ b\ +\ =$$

The stack now contains only one operation ; and thus, has the following form:

$$\boxed{\begin{array}{|c|}\hline ; \\ \hline \end{array}}$$

What we have in the stack – operation ; – means doing nothing. So, similarly to what we did with (), we can simply delete this from the stack. The stack is now empty.

Now, we see the symbol y of the expression

$$x \ = \ a \ + \ b;\ \underline{y} = Math.exp(x);$$

The symbol y represents numbers, so we add it to the postfix expression, which now takes the form

$$x\ a\ b\ +\ =\ y$$

The stack remains empty.

• Then, we see the assignment symbol =:

$$x \ = \ a \ + \ b;\ y \underline{=} Math.exp(x);$$

So, we push it into the stack. The stack now takes the form

$$\boxed{\begin{array}{|c|}\hline = \\ \hline \end{array}}$$

• The next operation symbol is $Math.exp$:

$$x \ = \ a \ + \ b; \ y \ = \ \underline{Math.exp}(x);$$

This operation symbol is pushed into the stack. As we discussed, everything after assignment has higher priority than assignment, so the assignment symbol is not popped.

$Math.exp$
$=$

• Then, we see the opening parenthesis:

$$x \ = \ a \ + \ b; \ y \ = \ Math.exp \ \underline{(} \ x \ );$$

Everything inside the parenthesis has to be performed first, so opening parenthesis has higher priority that everything before it. So, we push it into the stack, and we do not pop anything from the stack:

$($
$Math.exp$
$=$

• Next, we see the symbol x

$$x \ = \ a \ + \ b; \ y \ = \ Math.exp \ (\ \underline{x} \ );$$

which we add to the postfix expression:

$$x \ a \ b \ + \ = \ y \ x$$

• After that, we see the closing parenthesis:

$$x \ = \ a \ + \ b; \ y \ = \ Math.exp \ (\ x \ \underline{)};$$

So, we push it into the stack:

$)$
$($
$Math.exp$
$=$

As we mentioned earlier, when we have two parentheses in the stack – meaning $()$ – then both can be eliminated, so the stack takes the form

$Math.exp$
$=$

• Finally, we see the semicolon

$$x \ = \ a \ + \ b; \ y \ = \ Math.exp \ (\ x \ ) \ \underline{;}$$

As we mentioned earlier, the semicolon has lower priority that everything before it, so we push it into the stack, but before that, we pop everything from the stack and place it in the postfix expression. Now, the postfix expression takes the form

$$x \ a \ b \ + \ = \ y \ x \ Math.exp \ =$$

and the stack has only symbol – the semicolon:

;

A lone semicolon, as we mentioned earlier, can be deleted, so we get an empty stack.

• Thus, the expression

$$x \ a \ b \ + \ = \ y \ x \ Math.exp \ =$$

is the final postfix expression.

Let us summarize this:

x	$=$	a	$+$	b	$;$		y	$=$	$Math.exp$		$($	x	$)$	$;$	
x		a		b	$+$	$=$	y					x		$Math.exp$	$=$
	$=$		$+$		$=$			$=$	$Math.exp$		$($		$)$	$=$	
			$=$						$=$		$Math.exp$		$($		
											$=$		$Math.exp$		
													$=$		

To transform it into quadruples, let us apply the second algorithm.

• First, we see the three variables x, a, and b, so we place them into the stack. The stack how has the form

b
a
x

• Now, we see the symbol $+$:

$$x \ a \ b \ \underline{+} \ = \ y \ x \ Math.exp \ =$$

so we:

- pop b and a,
- add a quadruple
$$+ \quad a \quad b \quad r_1$$
and
- push r_1 into the stack:

r_1
x

• Next we see the assignment symbol:

$$x\ a\ b\ +\ =\!\!\!=\ y\ x\ Math.exp\ =$$

So, we:

- pop r_1 and x from the stack; and
- add a quadruple
$$= \quad x \quad r_1$$

For the assignment, there is no result, so we do not push anything into the stack. The stack is now empty.

• Then, we read variables y and x and push them into the stack:

x
y

• Next, we read the operation symbol $Math.exp$:

$$x\ a\ b\ +\ =\ y\ x\ \underline{Math.exp}\ =$$

This is a unary operation, so we only pop one symbol from the stack, not two, and the resulting quadruple will also lack the second operands; we will denote this by an underline _:

$$Math.exp \quad x \quad \_ \quad r_2$$

The stack now has the following form:

r_2
y

• Finally, we see the last symbol =, so we pop y and r_2 and get the last quadruple:

$$= \quad y \quad r_2$$

• As a result, we get the following compiled program:

$$+ \quad a \quad b \quad r_1$$

$$= \quad x \quad r_1$$

$$Math.exp \quad x \quad \_ \quad r_2$$

$$= \quad y \quad r_2$$

Let us summarize:

x	a	b	$+$	$=$	y	x	$Math.exp$	$=$
x	a	b	r_1		y	x	r_2	
	x	a	x			y	y	
		x						

Note. As before, we can eliminate unnecessary register variables.

Indeed, in the first quadruple

$$+ \quad a \quad b \quad r_1$$

we assign the sum $a + b$ to the register r_1, but then we immediately assign this value to x. Thus, we do not need r_1: we can directly assign $a + b$ to x:

$$+ \quad a \quad b \quad x$$

Similarly, in the third quadruple

$$Math.exp \quad x \quad \_ \quad r_2$$

we assign the value $Math.exp(x)$ to the register r_2, and then immediately assign this same value to y. So, we do not need r_2: we can directly assign the value $Math.exp(x)$ to y:

$$Math.exp \quad x \quad \_ \quad y$$

So, the resulting sequence of quadruples is:

$$+ \quad a \quad b \quad x$$

$$Math.exp \quad x \quad \_ \quad y$$

Practice. Try the same for some other simple program fragment.