The University of Birmingham
School of Computer Science
©Achim Jung and Uday Reddy

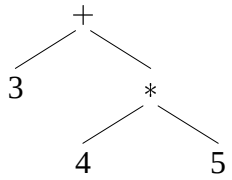
Handout 15

Using stacks: parsing of arithmetic expressions

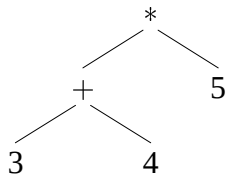
1. Let us pause for one lecture in our development of abstract data types to study an application of the abstract data type "stack": the parsing of arithmetic expressions. This is in fact one of the more complicated tasks of a compiler, the reason being that the language of arithmetic expressions was made for people, not machines, and so is full of shorthands and exceptions. Everything else in programming languages is usually highly regular and clearly structured.

2. **Expression trees.** Let us first study arithmetic expressions in general, before we look at the question how a machine might deal with them.

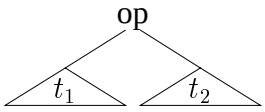
   Expressions are really **trees** although we do not typically write them in this way. Many questions about expressions become very simple and intuitive if one looks at them in the tree representation. For example, the expression $3 + 4 * 5$ as a tree takes the form

   which is clearly different from the tree for the expression $(3 + 4) * 5$:

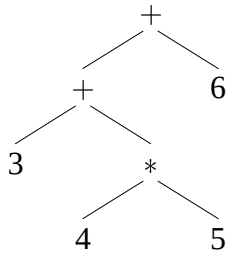
   Furthermore, the difference is expressed in the structure of the tree and it is not necessary to employ brackets. However, we don't have the space in our books, generally, to write every expression as a tree; instead we use a **linear representation** of trees. There are three possibilities for doing this. If we are given a tree of the general form

   where "op" is some operation symbol and t_1, t_2 are subexpressions, then we can write it down in the following ways

op t_1 t_2	**prefix** notation
(t_1) op (t_2)	**infix** notation
t_1 t_2 op	**postfix** notation

   where t_1 and t_2 are translated in the same way. A tree which consists of a value only is just written down as it is in any of these three representations. As an example, take the following tree:

Its linearisations are $++3*4\ 5\ 6$ (prefix), $(3+(4*5))+6$ (infix), and $3\ 4\ 5\ *+6+$ (postfix).

Note that we need brackets only for infix notation; the other two allow an unambiguous reconstruction from the linear notation back to the tree. As we all know, we can save some brackets in the infix notation by giving different operators different precedences (for the compiler writer this is another headache, unfortunately).

Prefix notation is sometimes called **Polish notation**, postfix is also known as **reversed Polish notation**.

3. **Evaluation of expressions.** If we are asked to **evaluate** a given expression, or if we want to find the assembler instructions necessary for the evaluation of an expression then by looking at the tree it is fairly easy to see how to proceed: One first evaluates the two subexpressions and then applies the operator to the resulting two values. The representation which supports this strategy is obviously postfix. If we are given an expression in infix notation (which is what we usually get from a user), then we first translate this into postfix and then apply an evaluator to the latter. For both processes we need a stack.

   An evaluator which takes a postfix expression and returns the result is a very simple program; it traverses the input token by token from left to right and for each token does the following: If the token is a number then this number is pushed onto a stack, if it is an operator symbol then the two top most values in the stack are popped, the operator is applied to them and the result is again pushed onto the stack.

   declare a stack s, initialized to empty;
   while there is more input **do** {
   read one item from the input, call it x;
   if (x is a number) **then** push x onto s;
   else {pop two numbers from the stack, call them b and a;
   apply the operator x to a and b, pushing the result onto s; }
   }
   pop one item from the stack and output it;

   If all goes well then the final result of the expression will be the only element on the stack at the end of the computation.

   How can verify that the algorithm works correctly? For this observe that postfix expressions are **inductively defined**, that is, one can give a finite **grammar** from which every expression can be generated in a unique way. The grammar is very simple:

   $$\mathrm{pexp} ::= number \quad | \quad \mathrm{pexp}\ \mathrm{pexp}\ \mathrm{op}$$

   It says that every postfix expression is either a number or of the form two postfix expressions followed by an operator symbol.

   We prove that given any stack, evaluating a postfix expression with the algorithm outlined above, will result in the stack containing one further entry, which is the value of the expression.

   base case If the expression consists just of a number then the algorithm pushes it onto the stack.

   induction step If the expression has the form $\mathrm{pexp}\ \mathrm{pexp}\ \mathrm{op}$, then the algorithm will start working on it from left to right. After processing the first subexpression it will, by the **Induction Hypothesis**, have added the value of that subexpression to the stack. Continuing the processing it will then add the value of the second subexpression to the stack, again by the Induction Hypothesis. Finally, the algorithm will process the operator symbol. In this case it will pop two values from the stack, apply the operator to them, and push the result back to the stack. We know that the two values popped are just the values of the two subexpressions, so the value that goes back onto the stack is precisely the overall value of the given expression.

   Note how this proof is based closely on the grammar for postfix expressions.

4. **Translation into postfix notation.** The more complicated task is to translate from infix to postfix. Since in infix we have the order "subexpression, operator, subexpression" and we want to bring the operator to the end, we need to save the operator somewhere and output it only after the second subexpression has been processed. Let us write down our first attempt at a translation algorithm:

   Procedure 1: (tentative)
   declare a storage cell s, initialized to empty string;
   while there is more input **do** {
   read one item from the input, call it x;
   if (x is a number) **then** output x;
   else if (s non-empty) **then** {
   output the contents of s;
   put x into s; }
   }
   if (s non-empty) **then** output s

The storage cell s is used to remember an operator until its second operand is output.

This works fine for expressions like $3 + 4 - 5 + 6 - 7$, where all operators have equal precedence. If we want to process expressions which contain operators of different precedence, such as $+$ and $*$, we need to be more careful as to when to output an operator.

If we are given an infix expression of the form "subexpression, operator1, subexpression, operator2, subexpression" then it depends on the precedences of operator1 and operator2 whether we should first evaluate the first three items or the last three items. In the first situation we transfer the first operator to the output and store the second one until the third subexpression is processed. If we are in the second situation (for example, if operator1 is $+$ and operator2 is $*$), then we need to store the second operator, output the third subexpression and then output operator2 and operator1. It follows that we need two places where to save operator symbols:

Procedure 2: (tentative)
declare storage cells $s0$ and $s1$, initialized to the empty string;
while there is more input **do** {
    read one item from the input, call it x;
    if (x is a number) **then** output x;
    else {
        if ($s1$ is non-empty) **then** output and clear $s1$;
        if ($s0$ is non-empty and precedence($s0$) $\geq$ precedence(x)) **then**
            output and clear $s0$;
        if ($s0$ is empty) **then**
            put x into $s0$;
        else put x into $s1$;
    }
}
if ($s1$ non-empty) **then** output $s1$;
if ($s0$ non-empty) **then** output $s0$;

In this algorithm, $s0$ holds the last lower precedence operator ($+$ or $-$) whereas $s1$ holds the last higher precedence operator ($*$ or $/$). When we get the next operator, it is necessarily going to be of lower or equal precedence than $s1$ (since we have only two precedence levels). Hence, we can blindly output $s1$. On the other hand, $s0$ may still be of lower precedence than the next operator. We output it only if it is of higher or equal precedence than the next one.

Note that the two variables $s0$ and $s1$ are being used as if they form a two-position stack. The variable $s1$ is used for storage ("pushing") only if $s0$ is filled. Moreover, we always clear $s1$ ("popping") before we look at $s0$.

If we have more than two precedence levels, then we need larger stacks for remembering other lower precedence operators. Moreover, brackets can be used build additional precedence levels within expressions. Brackets indicate that the subexpression contained between them should be evaluated before any operator on the outside is applied. For example, consider $3 * (4 + 5)$ or $3 * (4 + 2 * (6 + 5))$. Arbitrary levels of nested brackets would require a general stack to keep track of operators, not just a few locations. Our (final) algorithm:

Procedure 3:
declare a new stack s
while there is more input **do** {
    read one item from the input, call it x;
    case x **is**
        a number: output x;
        '(': push '(' onto s;
        ')': pop operators from the stack until '(' is found,
            copying the popped operators to the output and discarding '(';
        an operator: **if** (top(s) = '(') **then** push x onto s;
            else if (precedence(top(s)) $\geq$ precedence(x)) **then** {
                pop(s) until an operator of lower precedence than x
                    is found, or '(' is found, or stack is empty,
                    copying the popped operators to the output; }
            push x onto s;
    end case;
}
pop all operators in the stack, copying them to the output;

For example, parsing the expression $3 + 4 * (5 + 6 * (7 + 8))$ we go through the following stages:

remaining input	x	stack	output
$3 + 4 * (5 + 6 * (7 + 8))$			
$+4 * (5 + 6 * (7 + 8))$	3		
$4 * (5 + 6 * (7 + 8))$	+		3
$*(5 + 6 * (7 + 8))$	4	+	3
$(5 + 6 * (7 + 8))$	$*$	+	3 4
$5 + 6 * (7 + 8))$	(	+ $*$	3 4
$+6 * (7 + 8))$	5	+ $*$ (	3 4
$6 * (7 + 8))$	+	+ $*$ (	3 4 5
$*(7 + 8))$	6	+ $*$ (+	3 4 5
$(7 + 8))$	$*$	+ $*$ (+	3 4 5 6
$7 + 8))$	(	+ $*$ (+ $*$	3 4 5 6
$+8))$	7	+ $*$ (+ $*$ (	3 4 5 6
$8))$	+	+ $*$ (+ $*$ (	3 4 5 6 7
$))$	8	+ $*$ (+ $*$ (+	3 4 5 6 7
$)$	)	+ $*$ (+ $*$ (+	3 4 5 6 7 8
	)	+ $*$ (+ $*$	3 4 5 6 7 8 +
		+ $*$	3 4 5 6 7 8 + $*$ +
			3 4 5 6 7 8 + $*$ + $*$ +

Homework Exercise 8

(Submit via the pigeon hole, near the school office, next Tuesday, by 8pm.)

1. For each of the expressions, draw the expression trees and give the prefix and postfix equivalents:

   $$3 - 4 + 5 * 6 \qquad 3 + 4 - 5/6 * 7 \qquad 3 + (4 - 5)/(6 * 7)$$

2. (Similar to a question from a previous final exam.) Write an algorithm for translating postfix expressions into prefix notation. (There should be no parentheses in the output expressions.) *Hint*: Use a stack of expressions! Test it by translating the postfix expressions you created in Question 1.

3. Use the *first* algorithm in paragraph 4 to translate the expression $3 - 4 + 5 * 6$ into postfix. Is the resulting expression correct? Explain why not.

4. (From another old final exam.) Write an algorithm that takes as input a Java source code file and checks whether the following brackets appear balanced in it:

   $\{, \}; (,); [,]$

   (Before you start developing your procedure try to think of the different ways in which brackets can fail to be balanced. There are three of them!)

Announcement: Second Midterm Exam

The second midterm exam for Intro to Computer Science A will be held on Friday, 13th December, during the exercise class. The material covered for the exam includes handouts 9 through 15. Here is a simplified list of topics covered in these handouts:

1. Algorithms with loops
2. Arrays and linked lists
3. Stacks and queues

Anybody who has a conflict with the exam time should follow the instructions on the course web page for requesting a conflict exam.