Using Stacks: Algorithms for Infix, Postfix, and Prefix

• sections covered: 2.3 (except last subsection)
• topics:
  - definitions and examples
  - evaluating postfix expressions
  - converting infix to postfix expressions
Definitions and examples

- we consider simple arithmetic expressions with
  - binary operators: +, -, *, /, ^ (power), and parentheses
  - operands: single digit numbers (0, 1, ..., 9)
  - e.g.: 1+5*(4−2^3)
  - note: textbook uses "$" instead of "^", and sometimes letters instead of digits
Definitions and examples

• operator precedence (and associativity) is
  - lowest: +, - (left to right, e.g., 1-2-3 = (1-2)-3 )
  - middle: *, / (left to right, e.g., 1/2/3 = (1/2)/3 )
  - highest: ^ (right to left, e.g., 1^2^3 = 1^(2^3) )
• precedence determines order of evaluation:
  \[ 1+2*3^4 = 1+(2*(3^4)) \]
• precedence may be changed by parentheses:
  \[ ( (1+2) * 3 ) ^ 4 \]
Definitions and examples

• form with binary operators in between operands is called "infix", e.g., 1*2, (1*2)+3 = 1*2+3
• form with binary operator after operands is called "postfix", e.g., 12*, (12*)3+ = 12*3+
• form with binary operators before operands is called "prefix", e.g., *12, (+(*12)3 = +*123
Definitions and examples

- converting from/to infix/postfix/prefix:
  - set parentheses for each operator and its pair of operands
  - convert each operator separately (inside-out)
  - remove unnecessary parentheses
Definitions and examples

• example: infix 1+2*3 to postfix

  (1+(2*3))  add parentheses
  (1+(23*))  convert multiplication
  (1(23*)+)  convert addition
  123*+      remove parentheses
Definitions and examples

• further example: infix \((1+2) \times (3+4)\)
  - with parentheses: \(((1+2) \times (3+4))\)
  - in postfix: \(12+3\ 4+\star\)
  - in prefix: \(*+12+3\ 4\)

• one more: infix \(1^2 \times 3 – 4+5/6/(7+8)\)
  - paren.: \(((1^2 \times 3) – 4) + ((5/6)/(7+8)))\)
  - in postfix: \(12^\star3\ 4–5\ 6/\ 7\ 8+/+\)
  - in prefix: \(+–*^\star1\ 2\ 3\ 4\ //\ 56+78\)
Definitions and examples

• note: parentheses are never necessary for postfix and prefix, e.g.
  - infix: 1+( 2 * 3 ), postfix: 123*+, prefix: +1*23
  - infix: (1+2) * 3, postfix: 12+3*, prefix: *+123

• postfix and prefix are not just mirrored forms; however, algorithms for postfix and prefix are similar; therefore, we will focus on postfix only.
Evaluating postfix expressions

• example: $6 \ 2 \ 3 \ + \ - \ 3 \ 8 \ 2 \ + * \ 2 ^ 3 \ +$
  - scan from left to right: $6, 2, 3, +$
  - apply + to 2 and 3: $6, 5$
  - scan further: $6, 5, -$  
  - apply - to 6 and 5: $1$
  - scan further: $1, 3, 8, 2, /$
  - apply / to 8 and 2: $1, 3, 4$
  - scan further: $1, 3, 4, +$
Evaluating postfix expressions

- exam.: $623+(-382/+/^2*3^+)$, currently: $1, 3, 4, +$
  - apply $+$ to 3 and 4: $1, 7$
  - scan further: $1, 7, *$
  - apply $*$ to 1 and 7: $7$
  - scan further: $7, 2, ^$
  - apply $^$ to 7 and 2: $49$
  - scan further: $49, 3, +$
  - apply $+$ to 49 and 3: $52$
Evaluating postfix expressions

• note: scanned symbols are only added and removed at the end, i.e, we should use a stack!

• idea of the algorithm:
  - scan from left to right
  - push scanned operands on a stack
  - if an operator was scanned, apply it to the two topmost operands on the stack and replace them by the result
Evaluating a postfix expression

• algorithm:

operand_stack = empty stack

while (not end of input)

symbol = next input symbol

if (symbol is operand)
    push(operand_stack, symbol)

else

operand2 = pop(operand_stack)
operand1 = pop(operand_stack)
result = apply symbol to operand1 and operand2
push(operand_stack, result)

return pop(operand_stack)
Evaluating a postfix expression

• next: implementation in C
• using stackd.h and stackd.c which are adapted from stacki.h and stacki.c for doubles instead of ints (because the division of integers can result in non-integers).
• this code is not supposed to be reusable, but it should be able to handle any(!) input.
Evaluating a postfix expression

file: postfix.c (without comments)

```c
#include "stackd.h"
#include <assert.h>
#include <string.h>
#include <stdio.h>
#include <math.h>

#define MAX_SIZE_EXPRESSION 100
int main(void);
int evaluate_postfix(char *expression, double *result);
int evaluate_operator(int operator_symbol, double first_operand, double second_operand, double *result);
```

Evaluating a postfix expression

file: postfix.c (continued)

...  
int main(void) 
{
    char expression[MAX_SIZE_EXPRESSION];  
    int position = 0;  
    double value;  
    do 
    {  
        expression[position] = getchar();  
        position++;  
    } while (expression[position - 1] != '\n'  
              && position < MAX_SIZE_EXPRESSION);  
...
Evaluating a postfix expression

file: postfix.c (continued)

...  
expression[position - 1] = '\0';  
if (!evaluate_postfix(expression, &value))  
{  
    return 1;  
}  
printf("Expression \"%s\" evaluates to %g.\n",  
    expression, value);  
return 0;
}  
...
Evaluating a postfix expression

file: postfix.c (continued)

... 

int evaluate_postfix(char *expression, 
    double *result) 
{
    int position;
    stackd operand_stack;
    static char *digits = "0123456789";

    assert(NULL != result && NULL != expression);
    if (!stackd_init(&operand_stack, 0))
        return FALSE;
    ...

    ...
Evaluating a postfix expression

file: postfix.c (continued)

... for (position = 0;
     '\0' != expression[position]; position++)
{
    if (NULL !=
        strchr(digits, expression[position]))
    {
        if (!stackd_push(&operand_stack,
                        (double)(strchr(digits,
                        expression[position]) - &digits[0])))
            break;
    } ...

Evaluating a postfix expression

file: postfix.c (continued)

    ... 
    else 
    { 
        double first_operand, second_operand; 
        double value; 
        if (stackd_empty(&operand_stack)) 
            break; 
        second_operand = 
            stackd_pop(&operand_stack); 
        if (stackd_empty(&operand_stack)) 
            break; 
        first_operand = 
            stackd_pop(&operand_stack); 
    ...
Evaluating a postfix expression

file: postfix.c (continued)

...  
if (!evaluate_operator(
    expression[position], first_operand, 
    second_operand, &value))
    break;

    if (!stackd_push(&operand_stack, value))
        break;
}
} /* end else */
} /* end for */
...
Evaluating a postfix expression

file: postfix.c (continued)

...  
if ('\0' != expression[position]  
    || stackd_empty(&operand_stack))  
{
    printf("syntax error.\n");  
    stackd_deinit(&operand_stack);  
    return FALSE;
}
*result = stackd_pop(&operand_stack);  
...
Evaluating a postfix expression

file: postfix.c (continued)

    ... 
    if (!stackd_empty(&operand_stack))
    {
        printf("malformed expression.\n");
        stackd_deinit(&operand_stack);
        return FALSE;
    }
    stackd_deinit(&operand_stack);
    return TRUE;
    
    ...
Evaluating a postfix expression

file: postfix.c (continued)

...  
int evaluate_operator(int operator_symbol,
                       double first_operand, double second_operand,
                       double *result)
{
    assert(NULL != result);

    switch (operator_symbol)
    {
        case '+':
            *result = first_operand + second_operand;
            return TRUE;
        ...
    }
    ...

Spring 2004, EE 368 - Data Structures, Stacks
Evaluating a postfix expression

file: postfix.c (continued)

    ... 
    case '-': 
        *result = first_operand - second_operand; 
        return TRUE; 
    case '*': 
        *result = first_operand * second_operand; 
        return TRUE; 
    ...
Evaluating a postfix expression

file: postfix.c (continued)

...  

    case '/':
        if (0.0 == second_operand)
        {
            return FALSE;
        }
        else
        {
            *result = first_operand
            / second_operand;
            return TRUE;
        }
...
Evaluating a postfix expression

file: postfix.c (continued)

... 
... 
case '^
    if (first_operand <= 0.0)
    {
        return FALSE;
    }
    else
    {
        *result = pow(first_operand,
                        second_operand);
        return TRUE;
    
    }
...
Evaluating a postfix expression

file: postfix.c (continued)

    ...  
    default:
       return FALSE;
    } /* end switch */
    return FALSE; /* unreachable code */
}
Evaluating a postfix expression

• notes:
  - this implementation rejects all invalid expressions and evaluates all valid expressions (if there is enough memory for the stack, otherwise it fails)
  - we were able to reuse our stack code
  - this way of evaluating expressions is actually useful
Converting infix to postfix

• useful because evaluation of postfix is faster
• humans usually apply the rules of precedence to set parentheses, i.e., to determine the order of evaluation (and then build the postfix expression starting with the first operator), e.g., \(1 \times 2 + 3 = (1 \times 2) + 3\) leads to postfix \(1 \times 2 + 3\)
• how do we apply the rules of precedence?
Converting infix to postfix

• we can set parentheses around an operator whenever there is no operator with higher precedence to the left or the right of the operator
• if we scan from left to right, we can make sure that there is no operator with higher precedence to the left, but we still have to look to the right
• the order of operands does not change
Converting infix to postfix

- example: $1*2+3$
  - scan: $\ast, +$ (operators) and 12 (postfix)
  - precedence of $\ast$ is higher than $+$; thus,
    append $\ast$ to postfix: $+$ (ops.) and $12\ast$ (postfix)
  - scan further: $+$ (ops.) and $12\ast3$ (postfix)
  - there are no further operators, thus, postfix is $12\ast3+$
Converting infix to postfix

- second example: 1+2*3
  - scan: +, * (operators) and 12 (postfix)
  - precedence of + is lower than *; thus, scan further: +, * (operators) and 123 (postfix)
  - there are no further operators; thus, append operators starting with last scanned, i.e., postfix is 123*+
Converting infix to postfix

- third example: $1 + 2 \times 3^4 / 5 - 6$
  - scan: $+, \times$ (operators) and $1\,2$ (postfix)
  - precedence of $+$ is lower than $\times$; thus, scan further: $+, \times, ^$ (operators) and $1\,2\,3$ (postfix)
  - precedence of $\times$ is lower than $^$; thus, scan further: $+, \times, ^, /$ (ops.) and $1\,2\,3\,4$ (postfix)
  - precedence of $^$ is higher than $/$; thus, append $^$: $+, \times, /$ (ops.) and $1\,2\,3\,4\,^$ (postfix)
Converting infix to postfix

• third example: $1+2*3^4/5-6$
  - currently: $+, *, /$ (ops.) and $1234^*$ (postfix)
  - "precedence" of $*$ is higher than $/$; thus, append $*: +, /$ (ops.) and $1234^{*}*$ (postfix)
  - precedence of $+$ is lower than $/$; thus, scan further: $+, /, -$ (ops.) and $1234^{*}5$ (postfix)
  - precedence of $/$ is higher than $-$; thus, append $/: +, -$ (ops.) and $1234^{*}5/$ (postfix)
Converting infix to postfix

- third example: \(1+2 \times 3^4 / 5 - 6\)
  - currently: \(+, -, (\text{ops.})\) and \(1234^*5/\) (postfix)
  - "precedence" of + is higher than -; thus,
    append +: \(- (\text{ops.})\) and \(1234^*5/+\) (postfix)
  - scan: \(- (\text{ops.})\) and \(1234^*5/+6\) (postfix)
  - no more operators: \(1234^*5/+6-\) (postfix)
Converting infix to postfix

- third example again: 1+2 * 3 ^ 4 / 5 - 6

<table>
<thead>
<tr>
<th>symbol</th>
<th>operators</th>
<th>postfix</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>+</td>
<td>12</td>
</tr>
<tr>
<td>*</td>
<td>+, *</td>
<td>12</td>
</tr>
<tr>
<td>^</td>
<td>+, *</td>
<td>123</td>
</tr>
<tr>
<td>^</td>
<td>+, *, ^</td>
<td>123</td>
</tr>
<tr>
<td>4</td>
<td>+, *, ^</td>
<td>1234</td>
</tr>
<tr>
<td>/</td>
<td>+, *, ^, /</td>
<td>1234</td>
</tr>
<tr>
<td>+, /</td>
<td></td>
<td>1234</td>
</tr>
<tr>
<td>5</td>
<td>+, /</td>
<td>1234 ^ 5</td>
</tr>
</tbody>
</table>
Converting infix to postfix

• third example again: 1+2*3^4/5–6

<table>
<thead>
<tr>
<th>symbol</th>
<th>operators</th>
<th>postfix</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>+, /</td>
<td>1234^*5</td>
</tr>
<tr>
<td>–</td>
<td>+, /, –</td>
<td>1234^*5</td>
</tr>
<tr>
<td></td>
<td>+, –</td>
<td>1234^*5/</td>
</tr>
<tr>
<td></td>
<td>–</td>
<td>1234^*5/+</td>
</tr>
<tr>
<td>6</td>
<td>–</td>
<td>1234^*5/+6</td>
</tr>
</tbody>
</table>

1234^*5/+6–
Converting infix to postfix

- algorithm:
  set operator_stack to empty stack
  while (not end of input)
    symbol = next input
    if (symbol is operand)
      add symbol to postfix string
    else
      while (operator_stack not empty and top element has higher precedence than symbol)
        pop top element and add it to postfix string
      push symbol onto operator_stack
  while (operator_stack not empty)
    pop top element and add it to postfix string

Converting infix to postfix

- so far: only infix without parentheses
- ")" increases precedence of operators to the right and therefore delays operators on the stack
- "]" just "flushes" all operators on the stack until it finds its matching "("
Converting infix to postfix

- example with parentheses: \(1 - (2 + 3) \times 4\)

<table>
<thead>
<tr>
<th>symbol</th>
<th>operators</th>
<th>postfix</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>(</td>
<td>-, (</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>-, (</td>
<td>12</td>
</tr>
<tr>
<td>+</td>
<td>-, (+</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>-, (+</td>
<td>123</td>
</tr>
<tr>
<td>)</td>
<td>-</td>
<td>123+</td>
</tr>
<tr>
<td>*</td>
<td>-, *</td>
<td>123+</td>
</tr>
<tr>
<td>4</td>
<td>-, *</td>
<td>123+4</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>123+4*</td>
</tr>
</tbody>
</table>

123+4*–
Converting infix to postfix

• we use as function `has_precedence(stacktop_operator, symbol)` to determine whether the top element of the operator stack has precedence over the new input symbol
• precedence rules for "(" are chosen such that no top element has precedence over it and that "(" as a top element has no precedence over any other symbol
Converting infix to postfix

• precedence rules for ")" are chosen such that all top elements other than "(" have precedence
• since ")" is never put onto the operator stack, precedence of ")" as a top element is not defined
Converting infix to postfix

file: infix.c (no comments)

#include "stacki.h"
#include <assert.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#define MAX_SIZE_EXPRESSION 100

int main(void);
int convert_infix_to_postfix(char *infix_expr,
                              char *postfix_expr);
int has has_precedence(int operator1,
                        int operator2);
...
Converting infix to postfix

file: infix.c (continued)

... 

int main(void) {
    char infix_expr[MAX_SIZE_EXPRESSION];
    char postfix_expr[MAX_SIZE_EXPRESSION];
    int position = 0;
    do {
        infix_expr[position] = getchar();
        position++;
    } while (infix_expr[position - 1] != '\n' && position < MAX_SIZE_EXPRESSION);

...
Converting infix to postfix

file: infix.c (continued)

... 

infix_expr[position - 1] = '\0';

if (convert_infix_to_postfix(infix_expr, 
    postfix_expr))
{
    printf("Infix "\%s" in postfix: "\%s".\n", 
        infix_expr, postfix_expr);
}

return 0;

}"
Converting infix to postfix

file: infix.c (continued)

...  
int convert_infix_to_postfix(char *infix_expr,  
   char *postfix_expr)
{
    int infix_pos, postfix_pos;
    stacki operators_stack;
    int top_operator, symbol;

    if (NULL==infix_expr || strlen(infix_expr) + 1 > MAX_SIZE_EXPRESSION || NULL==postfix_expr  
       || !stacki_init(&operators_stack, 0))
      return FALSE;

    ...
Converting infix to postfix

file: infix.c (continued)

    ...    
    for (infix_pos = 0, postfix_pos = 0;
         '\0' != infix_expr[infix_pos];
        infix_pos++)
    {
        symbol = (int)infix_expr[infix_pos];
        if (NULL != strchr("0123456789", symbol))
        {
            postfix_expr[postfix_pos] = symbol;
            postfix_pos++;
        }     
    }    
    ...
else if (NULL != strchr("+-*/^()", symbol))
{
    while (!stacki_empty(&operators_stack))
    {
        top_operator = stacki_stacktop(
                        &operators_stack);
        if (!has_precedence(top_operator, symbol))
            break;
        stacki_pop(&operators_stack);
        postfix_expr[postfix_pos] = top_operator;
        postfix_pos++;
    }
    ...
}
Converting infix to postfix

file: infix.c (continued)

... 
if (')' != symbol)
{
    if (!stacki_push(&operators_stack, symbol))
    {
        printf("out of memory.\n");
        stacki_deinit(&operators_stack);
        return FALSE;
    }
}

...
Converting infix to postfix

file: infix.c (continued)

... else /* if (')' == symbol) */
{
    if (stacki_empty(&operators_stack)
        || '}' != stacki_stacktop(&operators_stack))
    {
        printf("missing \\n"; \\n"");
        stacki_deinit(&operators_stack);
        return FALSE;
    }
    stacki_pop(&operators_stack);
}...
Converting infix to postfix

file: infix.c (continued)

...
file: infix.c (continued)

...  
while (!stacki_empty(&operators_stack))  
{
    postfix_expr[postfix_pos] =
        stacki_pop(&operators_stack);
    postfix_pos++;  
}
postfix_expr[postfix_pos] = '0';
stacki_deinit(&operators_stack);
return TRUE;
}  
...
Converting infix to postfix

file: infix.c (continued)

... int has_precedence(int operator1, int operator2) {
    assert(NULL != strchr("+-*/^()", operator1) 
          && NULL != strchr("+-*/^()", operator2));
    if ('+' == operator1 || '-' == operator1) {
        if ('+' == operator2 || '-' == operator2 || ']' == operator2)
            return TRUE;
        else
            return FALSE;
    }
...
Converting infix to postfix

file: infix.c (continued)

...  
else if ('*' == operator1 || '/' == operator1)
{
    if ('^' == operator2 || '(' == operator2)
        return FALSE;
    else
        return TRUE;
}
...

Spring 2004, EE 368 - Data Structures, Stacks
Converting infix to postfix

file: infix.c (continued)

...  
else if ('^' == operator1)  
{  
    if ('^' == operator2 || '(' == operator2)  
        return FALSE;  
    else  
        return TRUE;  
}  
else if ('(' == operator1)  
    return FALSE;  
return FALSE; /* unreachable */
Converting infix to postfix

- code not intended for reuse
- code should handle all cases gracefully; however, for incorrect input it often generates incorrect output
- this is just an example for using stacks; there are better solutions for the problem of converting infix to postfix expressions.
Stacks - summary

• important stuff:
  - primitive operations (with preconditions) for stacks (the ADT)
  - implementation issues (stack objects, exceptional conditions, dynamic memory allocation, stacks for different element types, templates)

• less important: infix, postfix, and prefix