Stacks II

Adventures in Notation

The trouble with infix ...

- Rules for expression evaluation seem simple -- evaluate expression left to right, results of each sub-expression becoming operands to larger expression -- but …
- All operators are not created equal -- multiplication & division take precedence over addition & subtraction ...

The trouble with infix ...

- So it isn't really all that simple -- we must first scan for higher-precedence operations, and evaluate these first -- and that's not so easy to program -- so …
- In the calculator program, we rely on parentheses to tell us which operations to perform when -- hence the need for fully-parenthesized expression

Alternatives to infix -- prefix

- Prefix notation, a.k.a. Polish notation
- Operator precedes operands
  - infix expression: A + B
  - prefix expression: +AB
- Parentheses become unnecessary
  - infix expression: (A + B) * C
  - prefix expression: * + A B C

Converting from infix to prefix

- Write out operands in original order
- Place operators in front of their operands
- If there's a compound expression, the prefix expression may have two or more operators in a row
- If parentheses are not present, pay attention to precedence

Conversion examples

A + B + C >>>>>> + + A B C

A - B + C >>>>>> + - A B C

A + (B - C) >>>>> + A - B C

A * ((B + C) - D) / E >>> / * A - + B C D E

A + B * C / D >>>> + A / * B C D

A * B + C - D / E >>> - + * A B C / D E

Prefix evaluation

- scan left to right until we find the first operator immediately followed by pair of operands
- evaluate expression, and replace the “used” operator & operands with the result
- continue until a single value remains

Prefix Example

```
+ * / 4 2 3 9        // original expression
+ * 2 3 9            // 4/2 evaluated
+ 6 9                // 2*3 evaluated
15                   // 6+9 evaluated
```

Another example

* - + 4 3 5 / + 2 4 3	// original expression
* - 7 5 / + 2 4 3	// 4+3 evaluated
* 2 / + 2 4 3	// 7-5 evaluated
* 2 / 6 3	// 2+4 evaluated
* 2 2	// 6/3 evaluated
4	// 2*2 evaluated

Prefix summary

- Operands (but often not operators) same order as infix
- Expression designated unambiguously without parentheses
- Improvement on infix, but still not quite as simple to evaluate as one might wish -- have to deal with exceptions

Alternative II: Postfix

- Postfix is also known as reverse Polish notation -- widely used in HP calculators
- In postfix notation, operators appear after the operands they operate on
- As with prefix, operand order is maintained, and parentheses are not needed
- Postfix expression is ***not*** merely a reverse of the equivalent prefix expression

Postfix expression examples

- Simple expression:
  - Original Expression: A + B
  - Postfix Equivalent: A B +
- Compound expression with parentheses:
  - original: (A + B) * (C - D)
  - postfix: A B + C D - *
- Compound expression without parentheses:
  - original: A + B * C - D
  - postfix: A B C * + D -

Postfix expression evaluation

- Read expression left to right
- When an operand is encountered, save it & move on
- When an operator is encountered, evaluate expression, using operator & last 2 operands saved, saving the result
- When entire expression has been read, there should be one value left -- final result

Postfix evaluation using stack

- Postfix evaluation can easily be accomplished using a stack to keep track of operands
- As operands are encountered or created (through evaluation) they are pushed on stack
- When operator is encountered, pop two operands, evaluate, push result

Translating infix to postfix

- Postfix expression evaluation is easiest type to program
- Next task is to take an infix expression and translate it into postfix for evaluation
- Some basic assumptions:
  - all operations are binary (no unary negative)
  - expressions are fully parenthesized

Translating infix to postfix

- General method:
  - move each operator to the right of its corresponding right parenthesis
  - eliminate parentheses
- Example:

  (((A + B) * C) - (E * (F + G)))

  (((A B) + C) * (E (F G) +) *) -

  A B+ C * E F G + * -

Pseudocode for translation program

```
Do {
    if (left parenthesis) read & push
    else if (operand) read & write to output string
    else if (operator) read & push
    else if (right parenthesis)
            read & discard
            pop operator & write to output string
            pop & discard left parenthesis
} while (expression to read)
```

OK -- but what if expression ***isn't*** fully parenthesized?

- We have to fall back on the rules for expression evaluation we know & love
  - do expression in parentheses first
  - do other operations in order of precedence -- in case of tie, leftmost sub-expression wins
- Example: A - (B + C) * D - E
- order: 3 1 2 4
- Postfix: A B C + D * - E -

Algorithm for expression conversion

Do

if (left parenthesis) read & push

else if (operand) read & write to file

else if (arithmetic operator)

// continued on next slide

...

Conversion algorithm continued

while (stack not empty && stack.peek( ) != ‘(‘ &&
op precedence lower than stack.peek( ))
pop stack & write to file
read op
push op
else *// character should be ‘)’*
// next slide ...

Conversion algorithm continued

```
        read & discard right paren
        do
                pop stack & write to file
        while (stack.peek( ) != '(')
        pop & discard left paren
while (expression to read) // ends outer loop
while (stack not empty)
   pop & write to file
```