CSIS 3103

CH 3: Evaluating Arithmetic Expressions

Evaluating arithmetic expressions

Infix form
- Binary operators placed between operands

A computer normally scans an expression string in the order that it is input
- Easier to do with postfix expressions

Postfix Expressions

Postfix Expression	Infix Expression	Value
4 7 *	4 * 7	28
4 7 2 + *	4 * (7 + 2)	36
4 7 * 20 -	(4 * 7) - 20	8
3 4 7 * 2 / +	3 + ((4 * 7) / 2)	17

Advantages
- No need to group subexpressions in parentheses
- No need to consider operator precedence

Evaluating Postfix Expressions

Data Field	Attribute
`Stack<Integer> operandStack`	The stack of operands (`Integer` objects).
Method	**Behavior**
`public int eval(String expression)`	Returns the value of `expression`.
`private int evalOp(char op)`	Pops two operands and applies operator `op` to its operands, returning the result.
`private boolean isOperator(char ch)`	Returns **`true`** if `ch` is an operator symbol.

Evaluating Postfix Expressions

```
1. create an empty stack of integers
2. while there are more tokens
3.    get the next token
4.    if the first character of the token is a digit
5.       push the character on the stack
6.    else if the token is an operator
7.       pop the right operand off the stack
8.       pop the left operand off the stack
9.       evaluate the operation
10.      push the result onto the stack
11. pop the stack and return the result
```

Testing `PostfixEvaluator`

Read expressions and report results
- exercise each path by using each operator
- exercise each path through the method with different orderings and multiple occurrences of operators
- tests for syntax errors:
  - an operator without any operands
  - a single operand
  - an extra operand
  - an extra operator
  - a variable name
  - the empty string

Converting from Infix to Postfix

Data Field	Attribute
`private Stack<Character> operatorStack`	Stack of operators.
`private StringBuilder postfix`	The postfix string being formed.
Method	**Behavior**
`public String convert(String infix)`	Extracts and processes each token in `infix` and returns the equivalent postfix string.
`private void processOperator(char op)`	Processes operator `op` by updating `operatorStack`.
`private int precedence(char op)`	Returns the precedence of operator `op`.
`private boolean isOperator(char ch)`	Returns **`true`** if `ch` is an operator symbol.

Converting from Infix to Postfix

Next Token	Action	Effect on operatorStack	Effect on postfix
`w`	Append `w` to `postfix`.	[ ]	`w`
`-`	The stack is empty Push `-` onto the stack	[`-`]	`w`
`5.1`	Append `5.1` to `postfix`	[`-`]	`w 5.1`
`/`	`precedence(/)` > `precedence(-)`, Push `/` onto the stack	[`/` `-`]	`w 5.1`
`sum`	Append `sum` to `postfix`	[`/` `-`]	`w 5.1 sum`
`*`	`precedence(*)` equals `precedence(/)` Pop `/` off of stack and append to `postfix`	[`-`]	`w 5.1 sum /`

Converting from Infix to Postfix

Next Token	Action	Effect on operatorStack	Effect on postfix
`*`	`precedence(*)` > `precedence(-)`, Push `*` onto the stack	[`*` `-`]	`w 5.1 sum /`
2	Append 2 to `postfix`	[`*` `-`]	`w 5.1 sum / 2`
End of input	Stack is not empty, Pop `*` off the stack and append to `postfix`	[`-`]	`w 5.1 sum / 2 *`
End of input	Stack is not empty, Pop `-` off the stack and append to `postfix`	[ ]	`w 5.1 sum / 2 * -`

Converting from Infix to Postfix

Algorithm for Method `convert`

1. Initialize `postfix` to an empty `StringBuilder`.
2. Initialize the operator stack to an empty stack.
3. **`while`** there are more tokens in the infix string
4. Get the next token.
5. **`if`** the next token is an operand
6. Append it to `postfix`.
7. **`else if`** the next token is an operator
8. Call `processOperator` to process the operator.
9. **`else`**
10. Indicate a syntax error.
11. Pop remaining operators off the operator stack and append them to `postfix`.

Converting from Infix to Postfix

Algorithm for Method `processOperator`

1. **`if`** the operator stack is empty
2. Push the current operator onto the stack.
   `else`
3. Peek the operator stack and let `topOp` be the top operator.
4. **`if`** the precedence of the current operator is greater than the precedence of `topOp`
5. Push the current operator onto the stack.
   `else`
6. **`while`** the stack is not empty and the precedence of the current operator is less than or equal to the precedence of `topOp`
7. Pop `topOp` off the stack and append it to `postfix`.
8. **`if`** the operator stack is not empty
9. Peek the operator stack and let `topOp` be the top operator.
10. Push the current operator onto the stack.

Converting Expressions with Parenthesis

- Modify `processOperator` to push each opening parenthesis onto the stack as soon as it is scanned
- When a closing parenthesis is encountered, pop off operators until the opening parenthesis is encountered