ASSIGNMENT 4
An Interpreter for a Simple Programming Language

COMP-202, Summer 2010

Due: Tuesday, June 8th, 2010

1 Introduction

In the very first class we looked at the differences between **compiled** and **interpreted** languages. A compiled language we will use a program, called **compiler**, to translate our source code into an executable file. An interpreted language uses an **interpreter**, which reads in the source and directly performs the actions specified therein. We learned that `Java` is somewhat of a hybrid, because it uses both a compiler (`Javac`) and an interpreter (`Java`).

In this assignment we will make up a simple programming language, and implement an interpreter for it. Sometimes when solving tasks it is useful to come up with a simple language that can solve specific tasks. A language that is created for some specific purpose is called a **domain specific language**. `Java`, on the other hand, is a **general purpose language**. Our language is mostly there to evaluate math expressions, with some added features.

Note how `Java` programs are made up of classes, in which there are methods, in which there are statements, in which there can be expressions. An expression is the only of these constructs that directly computes a value. Our own language is going to be different: **Everything is an expression**. That means that there are no statements, only expressions, and all will compute a **value**. We model commands by having expressions with **side effects** (like the `++` im `java`). A language like this is called a **functional language**. A **pure functional language** do not allow side effects, so our language is not 'pure'. `Java` is considered an **object oriented language**, because the building blocks in that language are Objects. It is also considered **imperative**, because methods are made of statements, which are commands.

To make our language simpler, we will only use **prefix operators**. `Java` uses **infix operators** for mathematical expressions, meaning the operator is in between the operands: `a + b`, `x + y * z`. This will cause ambiguity, as we can see in the previous example - which operation gets executed first? In prefix notation the operator comes first, then the first, then the second operand. This makes the order of operations unambiguous, so we do not need to specify **operator precedence** or **order of operations**. But at the same time the expressions become very hard to read. The above examples would become `+ a b` and `+ x * y z`. We will also make our language easier by only having one type: `double`. All expressions will return a double.

Note that to evaluate an expression, we can write a simple interpreter which simply scans through the words (**tokens**) that come in at the terminal. Since everything is an expression, the first word will always be an operator (or an actual numerical value or variable). For example, if we see the `+` operator, we know there are two arguments. But both these arguments can again be expressions. This means we will use our simple interpreter again to evaluate the arguments - We will use our interpreter inside the interpreter. This assignment will be a first exercise in **recursion**, which will be used heavily. The **base cases** for our interpreter will be if the next token is a number, or a variable - because then we will not call the interpreter recursively.

Let's consider the example `+ - 1 3 * 2 3`. When scanning through this expression, we first encounter a `+`. Thus we will call our interpreter recursively on the remaining tokens. This call will encounter a `-` operator, and again invoke itself twice - once reading and returning a `1`, and once reading and returning a `3`. Taking the difference we will receive `-2` as the first argument for the `+` expression. The Interpreter will now need the second argument, and use itself again to find it. This call will encounter a `*` operator, which will call the interpreter twice, receiving a `2` and a `3`. Thus the result of the multiplication will be `6`. Our initial call can now finally compute the value of the sum, which is `4`. Again it is better to use a **"leap of faith"**, and assume that our interpreter will work, given that we have correct base cases (**literals**, **variables**), and correct recursive cases (**operations**).

We will build our simple language step by step, starting with just mathematical expressions. At every step we will add more features, which you should test thoroughly. You should also notice the differences from `Java`. After adding all specified features, you may implement the bonus. Use the `Do's and Don'ts` from before.

2 Specification

2.1 Evaluating Mathematical Expressions and Terminal (50 points)

Write a class called `Interpreter`, and add a method `evaluate`, taking a Scanner object, and returning a double. The method should read the tokens in the scanner and evaluate them as an expression, returning the result. Note that our language needs **white space** as **delimiters** (so entering something like `+3 4` does not work). If a token is not recognized, it should print an error and return the value `Double.NaN`, which is a special value meaning "Not a Number". You should also add a `main` method, which creates a scanner, and prompts the user for expressions, which are evaluated via the `evaluate` method. The results of the entered expressions should be printed back to the user (refer to the examples). Also add a static class variable of type `boolean`, called `exit`, initialized to `false`. The main method should keep reading expressions and returning the results until the exit flag is `true`. Hint: The base case is when a double value is entered. In order to check whether the next token at the scanner is a double, use the boolean method `hasNextDouble`, defined for scanners (refer to the documentation). In all other cases our token will be a `String`.

Implement the mathematical operators, constants and exit expression defined in the table below.

2.2 Variables (20 points)

Add three static class variables of type double, called `a`,`b`,`c`, initialized to 0. Add the expressions in the table, allowing both to read and write the variables. Note how assigning a variable is not a statement, but an expression. That is, assigning the variable will return the assigned value. Actually changing the value of the variable will only be the side effect of the expression. We only have the three variables in the whole language, so each has an two operations, to read it and to write it.

2.3 If-Then-Else Expressions (20 points)

In our language, if-then-else constructs are expressions. That means they evaluate the condition (1st operand), and return either the 2nd, if it's "true", or the 3rd operand, if it's "false". Since evaluating an expression may have side effects, we need a way to skip either the 2nd operand or the 3rd operand without evaluating it. Thus we need a new method `skip`, taking also a Scanner, but returning a `String`. Just like the `evaluate`, method it will recursively step through an expression, but it will not perform any of the computations. Instead it will just skip the tokens that form the expression. The skipped tokens get returned as a `String`, with spaces added in between as delimiters. Hint: Previously we needed an `if` for every possible command. Now all operands that have the same number of arguments will always do the same thing.

Since we have only doubles, we need a way to denote true and false, without using booleans. We use the convention (used in some programming languages, but not `Java`) that any nonzero value represents "true", and `0` represents "false". Implement the listed **logical operators**, which will not return true or false, but 1 (representing "true") or 0 (representing "false").

2.4 Functions (10 points)

So far we can only have expressions, but we would like methods as well. Since we are in a functional language, we call them functions here, like we would in math. We only have two functions, `f` and `g`. We will store functions as Strings (static class variables `f`, `g`), allowing us to evaluate them later. Note that you can create `Scanner` object not only from `System.in`, but also from any `String` (see the documentation), allowing us to read tokens from it. The `f=` and `g=` operands will both read in the following expression, but not evaluate them. Instead they will be stored as a `String`, explaining why `skip` returns a String.

All functions will always take one argument. Inside the function, this parameter will be referred to by x. But how will the `evaluate` method know what value x has? We cannot create a global variable for it, because then our functions could not call each other or themselves (because we can only have one active call - another would overwrite x). In order to do this, we will add a second parameter of type double to the `evaluate` method. When calling a function, we will first evaluate its argument, and assign it to that second double parameter, allowing us to read it. Sometimes when coding, we will need to make these major changes affecting a lot of the code. This process is called **refactoring**.

Language Specification

The following specifies our language. Note that since everything is an expression, there are always return values. An expression may or may not have side effects. Note that side effects get ignored when evaluating if statements (on the non-executing branch), and when defining functions.

group	Syntax	Side Effect	Returned Value
literal	`<double literal>`	-	the value of the literal
binary operations	`+ <e1> <e2>`	-	$e_1 + e_2$
	`- <e1> <e2>`	-	$e_1 - e_2$
	`* <e1> <e2>`	-	$e_1 \cdot e_2$
	`/ <e1> <e2>`	-	$\frac{e_1}{e_2}$
	`^ <e1> <e2>`	-	$e_1^{e_2}$
functions	`sin <e1>`	-	$\sin(e_1)$
	`cos <e1>`	-	$\cos(e_1)$
	`exp <e1>`	-	e^{e_1}
	`log <e1>`	-	$\ln(e_1)$
	`sqrt <e1>`	-	$\sqrt{e_1}$
constants	`pi`	-	π
exit	`exit`	sets the `exit` flag to `true`	0
writing variables	`a= <e1>`	assigns e_1 to a	e_1
	`b= <e1>`	assigns e_1 to b	e_1
	`c= <e1>`	assigns e_1 to c	e_1
reading variables	`a`	-	a
	`b`	-	b
	`c`	-	c
logical operators	`== <e1> <e2>`	-	$\begin{cases} 1 \text{ if } e_1 = e_2, \\ 0 \text{ if } e_1 \neq e_2 \end{cases}$
	`< <e1> <e2>`	-	$\begin{cases} 1 \text{ if } e_1 < e_2, \\ 0 \text{ if } e_1 \geq e_2 \end{cases}$
if-then-else expression	`if <e1> <e2> <e3>`	skips side effects of e_2 or e_3	$\begin{cases} e_2 \text{ if } e_1 \neq 0, \\ e_3 \text{ if } e_1 = 0 \end{cases}$
writing functions	`f= <e1>`	assigns the text of e_1 to f	0
	`g= <e1>`	assigns the text of e_1 to g	0
calling functions	`f <e1>`	-	$f(e_1)$
	`g <e1>`	-	$g(e_1)$
accessing parameters	`x`	-	$\begin{cases} x \text{ if assigned} \\ 0 \text{ if unassigned} \end{cases}$
Bonus	`{ <e1> <e2> .. <en> }`	-	e_n

3 bonus

3.1 Block Expressions (10 points)

Sometimes we want to execute a couple of expressions with side effects, but discared their results. For this we will make a "block expression" (see table). It allows us to group multiple expressions together. They will all get computed, but only the value of the last expression gets returned. Note that in order to do that we have to keep evaluating expressions, continuously checking whether the net token might be a } . We can do this by using the Scanner method `hasNext(String pattern)`. As with `hasNextDouble()`, this method does not skip the next token, even if it matches the pattern.

4 Example Output

4.1

```
>> 3
result: 3.0
>> pi
result: 3.141592653589793
>> + 1 2
result: 3.0
>> + * + 1 2 3 4
result: 13.0
>> - + * / ^ 1 2 3 4 5 6
result: 0.33333333333333304
>> log exp sqrt ^ * 2 + ^ cos pi 2 ^ sin pi 2 2
result: 2.0
>> blarbigh
unrecognized expression: blarbigh
result: NaN
>> + - foo bar 9
unrecognized expression: foo
unrecognized expression: bar
result: NaN
>> exit
result: 0.0
```

4.2

```
>> a= .5
result: 0.5
>> + a pi
result: 3.641592653589793
>> a=4
unrecognized expression: a=4
result: NaN
```

4.3

```
>> a= .5
result: 0.5
>> if == a .5 * a a sqrt a
result: 0.25
>> if 1 2 3
result: 2.0
>> if 0 2 3
result: 3.0
>> a= 1
result: 1.0
>> if a a= 3 b= 4
result: 3.0
>> a
result: 3.0
>> b
result: 0.0
```

4.4

4.4.1 Factorial

```
>>  f= if < 0 x * x f - x 1 1
result: 0.0
>> f 1
result: 1.0
>> f 2
result: 2.0
>> f 3
result: 6.0
>> f 4
result: 24.0
>> x
result: 0.0
```

4.4.2

In the following, we set up g such that it returns "true" if the argument is an integer.

```
>>  g= if < 0 x g - x 1 if == x 0 1 0
result: 0.0
>> g 4.5 g 9 g 1.5 g 3.5 g 2
result: 0.0
>> result: 1.0
>> result: 0.0
>> result: 0.0
>> result: 1.0
>> g 5
result: 1.0
```

4.4.3

The following returns whether a is a prime number, given a as an argument. Note that we need the g from above.

```
f= if == a x if f - x 1 0 1 if < 1 x + f - x 1 g / a x 0
>> a= 97
result: 97.0
>> f a
result: 1.0
>> a= 96
result: 96.0
>> f a
result: 0.0
```

4.5 Bonus

```
>> { a= + 0 1 b= + a 1 c= b 1 }
result: 1.0
>> g= { a= x b= * x x 1 }
result: 0.0
>> g 2
result: 1.0
>> a
result: 2.0
>> b
result: 4.0
>>
```