Expressions, Functions, and the Substitution Model

In Haskell, computation is done on *expressions*. You have seen expressions before in Java. Here are some examples:

```
3 + 4

abs -9.0

sqrt (3 + 6)
```

What makes something an *expression*? An expression has a *value* and that value has some *type*. Look at the values and types of the previous expressions:

```
3 + 4   =>  7 : Int

abs -9.0  =>  9.0 : Float

sqrt (3 + 6)  => 3.0 : Float
```

We can define *functions* which takes parameters of some type and return the value of an expression that uses the parameters. We usually declare the function by specifying the types of the parameters as well as the type of the returned value. Consider the function *square* below. It takes one parameter x, which is of type Float and returns the value of the expression $x * x$ which is also a Float.

```
square :: Float -> Float
square x = x * x
```

The following function *hypotenuse* takes two parameters a and b and returns the expression that returns the square root of the sum of the squares of the parameters. We will discuss later why the types are declared in this strange way. but the last Float refers to the return value and everything before that refers to the types of the parameters. Observe that we are using the function *square* that we defined previously.

```
hypotenuse :: Float -> Float -> Float
hypotenuse a b = sqrt (square a + square b)
```

Now that we have declared some functions, suppose that we wish to use our function *hypotenuse* to compute the value of the expression:

```
hypotenuse (3 + 2) (3 * 4)
```

To figure out by hand how the result of this expression, we would simply *substitute* the expression definition for the function hypotenuse and replace its parameters with the values of its argument expressions, in this case (3+2) for a and (3*4) for b as follows:

Expression	Substitution explanation
$hypotenuse(3+2)(3*4)$	substitute into the body of *hypotenuse*
$sqrt(square(3+2)+square(3*4))$	evaluate the arguments of *square*
$sqrt(square5+square12)$	substitute in the body of *square*
$sqrt(5*5+12*12)$	evaluate the argument of *sqrt*
$sqrt(169)$	evaluate *sqrt*
13.0	

We will find this *substitution model* quite useful as expressions get more complex.

Application: Manipulating Words and Sentences

We will consider an application of manipulating words and sentences. We will give both words and sentences the type *Language* and have two functions *word* and *sent* which take a *String* and convert it to type *Language*. We have lots of functions we can use to manipulate this type:

Function	Explanation
firstItem :: *Language* -> *Language*	gives the first letter or word of a word or sentence
lastItem :: *Language* -> *Language*	gives the last letter or word of a word or sentence
butFirst :: *Language* -> *Language*	gives everything BUT the first letter or word of the word or sentence
butLast :: *Language* -> *Language*	gives everything BUT the last letter or word of the word or sentence
item :: *Int* -> *Language* -> *Language*	gives the *nth* letter or word from the word or sentence
count :: *Language* -> *Int*	gives the number of letters or words in the word or sentence
(+++) :: *Language* -> *Language* -> *Language*	concatenate two *Language* objects together
empty :: *Language* -> *Bool*	determines if a word or sentence is empty
member :: *Language* -> *Language* -> Bool	determines if a letter or word is contained in a word or sentence
wordToSent :: *Language* -> *Language*	converts a word to a sentence of one word
sentToWord :: *Language* -> *Language*	converts a sentence to a single word

Consider the following example expressions with their values and types:

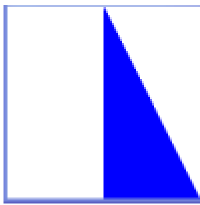
```
word "computer" +++ word "s"   =>  computers : Language

sent "the answer is" +++ word "42"   =>  [the answer is 42] : Language

count(butFirst (word "dogs")) => 3 : Int
```

Consider the following function definitions. Can you figure out what they do?

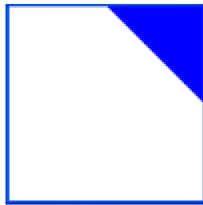
```
addS :: Language -> Language
addS w = w +++ word "s"

thirdPerson :: Language -> Language
thirdPerson verb = sent "she" +++ addS verb
```

Application: Quilts

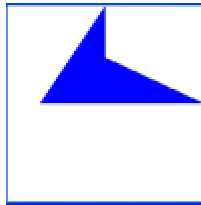
Now we will look at building expressions which are not composed of numbers or Language values. Instead we will be manipulating quilt pieces which are of type *Image*. We have four basic block values which we are given:



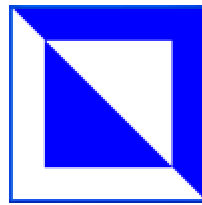
(a) testBB



(b) cornerBB



(c) novaBB



(d) rcrossBB

We have two functions to manipulate the images:

Function	Explanation
quarterTurnRight :: *Image* -> *Image*	rotates the image by a quarter turn right
stack :: *Image* -> *Image* -> *Image*	stacks two images (of equal width)

We need to use the function *draw* to display an image. Try to figure out what each of these expressions will display:

```
draw (stack (quarterTurnRight testBB) rcrossBB)

draw (quarterTurnRight (stack testBB testBB))
```