Chapter 1

Problem Solving, Programming, and Calculation

Introduction

- Be sure to *understand your problem well.*
- Find a solution that is *correct*. But there may be several solutions that trade off efficiency (in time, space, number of processors, etc.) and clarity.
- In this text *Haskell* is used to express solutions to problems of many kinds.
- Haskell is a *functional language* quite unlike most conventional programming languages (such as C, C++, Java, Visual Basic, etc.).
- To best understand Haskell, understand the concept of *computation by calculation*.

Computation by Calculation

- Computation by calculation is a simple concept that everyone should be familiar with, since it's not unlike ordinary arithmetic calculation. For example:
  ```
  3 * (9 + 5)
     ➔ 3 * 14
     ➔ 42
  ```
- However, since we want computers to perform these tasks, we are also interested in issues such as efficiency:
  ```
  3 * (9 + 5)
     ➔ 3*9 + 3*5
     ➔ 27 + 3*5
     ➔ 27 + 15
     ➔ 42
  ```
- Same answer, but the former was *more efficient* than the latter, since it took a fewer number of steps.

Abstraction

- We are also interested in *abstraction:* the process of recognizing a repeating pattern, and capturing it succinctly in one place instead of many.
- For example:

  `3*(9+5)      4*(6+2)      7*(8+1)      ...`

  This repeating pattern can be captured as a *function:*

  `easy x y z = x*(y+z)`

  Then each instance can be replaced by:

  `easy 3 9 5    easy 4 6 2    easy 7 8 1    ...`

- We can also perform calculations with *symbols*. For example, we can *prove* that **`easy a b c = easy a c b`**:

```
easy a b c
➔ a*(b+c)           { unfold }
➔ a*(c+b)           { commutativity of + }
➔ easy a c b        { fold }
```

Expressions, Values, and Types

- The objects on which we calculate are called *expressions.*
- When no more unfolding (of either a primitive or user-defined function) is possible, the resulting expression is called a *value.*
- Every expression (and therefore every value) has a *type.* (A type is a collection of expressions with common attributes.)
- We write **`exp :: T`** to say that expression **`exp`** has type **`T`**.
- Examples:
  - Atomic expressions:
    `42 :: Integer`, **`'a' :: Char`**, **`True :: Bool`**
  - Structured expressions:
    `[1,2,3] :: [Integer]` - a *list* of integers
    `('b',4) :: (Char,Integer)` – a *pair* consisting of a character and an integer
  - Functions:
    `(+) :: Integer -> Integer -> Integer`
    `easy :: Integer -> Integer -> Integer -> Integer`

Abstraction

- Our derivation of the function **`easy`** is a good example of the use of the *abstraction principle*: separating a repeating pattern from the particular instances in which it appears. In particular, the example demonstrates *functional abstraction.*
- *Naming* is an even simpler kind of abstraction:

```
let pi = 3.14159
in 2*pi*r1 + 2*pi*r2
```

- *Data abstraction* is the use of data structures to store common values on which common operations may be applied in an abstract manner.
- The "circle areas" example from the text demonstrates all three kinds of abstraction.

"Circle Areas" Example

- Original program:

```
totalArea r1 r2 r3 = pi*r1^2 + pi*r2^2 + pi*r3^2
```

- A more abstract program:

```
totalArea r1 r2 r3 =
  listSum [circArea r1, circArea r2, circArea r3]

listSum []     = 0
listSum (a:as) = a + listSum as

circArea r = pi*r^2
```

- The new program is longer than the old – in what ways is it better?
  - The code for the area of a circle has been isolated (using functional abstraction), thus minimizing errors and facilitating change and reuse.

Proof by Calculation

- Proof that the new **`totalArea`** is equivalent to the old:

```
listSum [circArea r1, circArea r2, circArea r3]
```
➔ { unfold listSum }
```
circArea r1 + listSum [circArea r2, circArea r3]
```
➔ { unfold listSum }
```
circArea r1 + circArea r2 + listSum [circArea r3]
```
➔ { unfold listSum }
```
circArea r1 + circArea r2 + circArea r3 + listSum []
```
➔ { unfold listSum }
```
circArea r1 + circArea r2 + circArea r3 + 0
```
➔ { unfold circArea (three places) }
```
pi*r1^2 + pi*r2^2 + pi*r3^2 + 0
```
➔ { simple arithmetic }
```
pi*r1^2 + pi*r2^2 + pi*r3^2
```