Haskell: Lists

CS F331 Programming Languages
CSCE A331 Programming Language Concepts
Lecture Slides
Friday, February 24, 2017

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

Review
PL Categories: Functional PLs — Background

Functional programming (**FP**) is a programming style that generally has the following characteristics.

- Computation is considered primarily in terms of the *evaluation* of functions (as opposed to *execution* of code).
- Thus, functions are a primary concern. Rather than considered as repositories for code, functions are values to be constructed and manipulated.
- Side effects & mutable data are avoided. The only job of a function is to return a value.
  - A **side effect** occurs when a function alters data, and this alteration is visible outside the function.
  - **Mutable** data is data that can be altered.

A **functional programming language** is a PL designed to support FP well.

Review
PL Categories: Functional PLs — Typical Characteristics

A typical functional programming language has the following features/characteristics.

- It has first-class functions.
- It offers good support for higher-order functions.
  - A **higher-order function** is a function that acts on functions.
- It offers good support for recursion.
- It has a preference for immutable data.

A **pure** functional PL goes further, and does not support mutable data at all. There are no side effects in a pure functional PL.

Introduction to Haskell — Characteristics: Type System

Haskell is a pure functional PL. It has first-class functions and good support for higher-order functions.

Haskell has a sound static type system with sophisticated type inference. So typing is largely inferred, and thus implicit; however, we are *allowed* to use manifest typing, if we wish.

Haskell's type-checking standards are difficult to place on the nominal-structural axis.

Haskell has few implicit type conversions. New implicit type conversions cannot be defined.

Introduction to Haskell — Characteristics: Flow of Control

Haskell implementations are required to do **tail call optimization** (**TCO**). This means that the last operation in a function is not implemented via a function call, but rather as the equivalent of a goto, never returning to the original function.

Iteration is difficult without mutable data. And, indeed, Haskell has no iterative flow-of-control constructs. It uses recursion instead, with tail recursion preferred. The latter will generally be optimized using TCO.

Haskell has **significant indentation**. Indenting is the usual way to indicate the start & end of a block.

By default Haskell does **lazy evaluation**: expressions are not evaluated until they need to be. C++, Java, and Lua do the opposite, evaluating as soon as an expression is encountered; this is **eager evaluation** (or **strict evaluation**).

Review
Haskell: Functions — Basic Syntax [1/3]

The material for this topic is also covered in a Haskell source file, which is extensively commented.

See **`func.hs`**.

Haskell expression: stream of words separated by blanks where necessary. Optional parentheses for grouping.

- Each line below is a single Haskell expression. Type it at the GHCi prompt and press <Enter> to see its value.

```
2+3
(2+3)*5
reverse "abcde"
map (\ x -> x*x) [1,2,3,4]
```

Review
Haskell: Functions — Basic Syntax [2/3]

Comments

- Single line, two dashes to end of line: `-- …`
- Multi-line, begin with brace-dash, end with dash-brace: `{- … -}`

Identifiers begin with letter or underscore, and contain only letters, underscores, digits, and **single quotes** (`'`).

- *Normal* identifiers begin with lower-case letter or underscore. These are used to name variables and functions.

```
myVariable
my_Function'_33
```

- *Special* identifiers begin with UPPER-CASE letter. These are used to name types, modules, and constructors.

```
MyModule
```

Review
Haskell: Functions — Basic Syntax [3/3]

Define a variable by giving its name, and equals sign (=) and an expression for the value.

```
ab'c = 7 * (3 + 2)
```

A variable definition is *not* an expression in Haskell.

The above is legal in a Haskell source file. At the GHCi prompt it must be preceded by **`let`**.

```
let ab'c = 7 * (3 + 2)
```

Review
Haskell: Functions — Defining Functions & Operators [1/3]

To define a Haskell function, write what looks like a call to the function, an equals sign, and then an expression for what the function returns.

```
addem a b = a+b
```

We can also define new operators. Infix binary operators have names consisting of special characters. They are defined similarly.

```
a +$+ b = 2*a + b
```

Haskell: Functions — Defining Functions & Operators [2/3]

Function definitions use **pattern matching**. Define a function differently for different patterns. The first matching pattern is the one used.

Here is a factorial function.

```
factorial 0 = 1
factorial n = n * factorial (n-1)
```

Review
Haskell: Functions — Defining Functions & Operators [3/3]

We can use a regular function as an infix operator by surrounding its name with backquotes (`). Having defined function addem, try the following at the GHCi prompt.

```
2 `addem` 3
```

And we can use an operator as a regular function by surrounding its name with parentheses. Having defined **+$+**, try the following at the GHCi prompt.

```
(+$+) 5 7
```

Haskell: Functions — Local Definitions

Use **where** to introduce a block (indent!) of local definitions.

```
plus_minus times a b c d = a_plus_b * c_minus_d where
    a_plus_b = a + b
    c_minus_d = c - d
```

Local-definition blocks can be nested.

```
twiceFactorial n = twice (factorial n) where
    twice k = two*k where
        two = 2
    factorial 0 = 1
    factorial curr = curr * factorial prev where
        prev = curr-1
```

Review
Haskell: Functions — Function Application & Currying

Currying mean simulating a multiparameter function using a single parameter function that returns a function.

For example, our function **addem** really takes one parameter. It returns a function that adds that parameter to something. The following are the same:

```
addem 2 3    -- Returns 5
(addem 2) 3  -- Returns 5
```

We can give the intermediate function a name.

```
add2 = addem 2
add2 3  -- Returns 5
```

Haskell: Functions — Higher-Order Functions

A **higher-order function** is a function that acts on functions.

```
rev f a b = f b a

sub a b = a - b
rsub = rev sub

sub 5 2   -- Returns 3
rsub 5 2  -- Returns -3
```

Review
Haskell: Functions — Lambda Functions

A **lambda function** (or **lambda expression**) is a kind of expression whose value is a function.

- The name comes from the **lambda calculus**, a mathematical system in which everything is a function. In this system, an unnamed function is introduced using the Greek letter lambda (λ).
- Haskell uses a backslash (\) since it looks a bit like a lambda.

```
square x = x*x
square' = \ x -> x*x  -- Using lambda function;
                      --  same as above

-- Alternate definitions for addem
addem' = \ x y -> x+y
addem'' a = \ y -> a+y
addem''' = \ x -> (\ y -> x+y)
```

Haskell: Lists
Lists & Tuples [1/4]

A statically typed PL will typically support two ways of aggregating multiple data items into a single collection:

- A collection of an arbitrary number of data items, all of the same type. Example. C++ **`vector`**, **`list`**, **`deque`**.
- A collection of a fixed number of data items, possibly of different types. Example. C++ **`tuple`**, **`struct`**.

Haskell supports the above two categories as well, in the form of **lists** and **tuples**.

For code, see **`list.hs`***.*

Lists & Tuples [2/4]

A Haskell **list** holds an arbitrary number of data items, all of the same type. A list literal uses brackets and commas.

```
[]                    -- Empty list
[2, 5, 3]             -- List of three Integer values
["hello", "there"]    -- List of two String values
[[1], [], [1,2,3,4]]  -- List of lists of Integer
[1, [2, 3]]           -- ERROR; types differ
```

Haskell lists can be infinite.

```
[1, 3 ..]  -- List of ALL nonnegative odd Integers
```

Haskell: Lists
Lists & Tuples [3/4]

The type of a list is written as the item type in brackets.

This represents the GHCi prompt.

```
> :t [True, False]
[True, False] :: [Bool]
```

Lists with different lengths can have the same type.

```
> :t [False, True, True, True, True, False]
[False, True, True, True, True, False] :: [Bool]
```

Lists & Tuples [4/4]

A Haskell **tuple** holds a fixed number of data items, possibly of different types. A tuple literal uses parenthesis and commas.

```
(2.1, 1.2, "hello", True)  -- Tuple: Double, Double,
                           --  String, Bool
```

Haskell tuples cannot be infinite.

The type of a tuple is written as if it were a tuple of types.

```
> :t (2.1, True)
(2.1, True) :: (Double, Bool)
```

Tuples with different numbers of items always have different types.

List Primitives [1/2]

A **primitive** operation (or simply **primitive**) is one that other operations are constructed from.

Haskell has three list primitives.

1. Construct an empty list.

```
[]
```

2. **Cons**: construct a list given its first item and a list of other items. Uses the infix colon (`:`) operator.

```
[5, 2, 1, 8]
5:[2, 1, 8]  -- Same as above
5:2:1:8:[]   -- Also same; ":" is right-associative
```

Continued …

Haskell: Lists
List Primitives [2/2]

(three Haskell list primitives, continued)

3. Pattern matching for lists.

```
ff [] = 3       -- Value of ff for an empty list
ff (x:xs) = 4   -- Value of ff for a nonempty list
```

Read "**x:xs**" as "**x** and some **xs** (plural)". This is a common convention.

Above, the parentheses around **(x:xs)** are required due to precedence: function application has very high precedence.

```
gg [a] = 17         -- Value of gg for a 1-item list
gg [a, b, c] = 19   -- Value of gg for a 3-item list
```

Haskell: Lists
Other List Syntax — Strings & Ranges

A Haskell **`String`** is a list of characters (**`Char`** values).

```
['a', 'b', 'c']
"abc"  -- Same as above
```

Use ". ." to construct a list holding a range of values. There are exactly four ways to do this.

```
[1..10]     -- Same as [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1,3..10]   -- Same as [1, 3, 5, 7, 9]
[1..]       -- Infinite list: [1, 2, 3, 4, 5, 6, 7, 8, …]
[1,3..]     -- Infinite list: [1, 3, 5, 7, 9, 11, …]
```

These four are wrappers around **`enumFromTo`**, **`enumFromThenTo`**, **`enumFrom`**, and **`enumFromThen`**, respectively.

Other List Syntax — List Comprehensions [1/2]

You have probably seen the mathematical notation known as a **set comprehension** (or *set-builder notation*). Here is an example.

$$\{\ xy \mid x \in \{3, 2, 1\} \text{ and } y \in \{10, 11, 12\}\ \}$$

The above is read as, "The set of all *xy* for *x* in the set {1, 2, 3} and *y* in the set {10, 11, 12}."

A number of PLs, including Haskell, have a construct based on this idea: the **list comprehension**. Here is a Haskell example.

```
[ x*y | x <- [3, 2, 1], y <- [10, 11, 12] ]
```

This deals with Haskell lists instead of sets, but is otherwise very similar to the above set comprehension.

Haskell: Lists
Other List Syntax — List Comprehensions [2/2]

The syntax of a Haskell list comprehension is as follows. Brackets enclose the following:

- An expression.
- Then a vertical bar (`|`).
- Then a comma-separated list of two kinds of things:
  - *var* `<-` *list*
  - Expression of type `Bool`

Here are some examples:

```
> [ x*y | x <- [3, 2, 1], y <- [10, 11, 12] ]
[30,33,36,20,22,24,10,11,12]
> [ x*x | x <- [1..6] ]
[1,4,9,16,25,36]
> [ x | x <- [1..20], x `mod` 2 == 1]
[1,3,5,7,9,11,13,15,17,19]
```

Lists & Recursion [1/3]

When writing a function that takes a list, it is very common to have two cases.

- One case handles the empty list: **`[]`**.
- The other case handles nonempty lists. Remember that a pattern like **`a:as`** matches nonempty lists.

```
isEmpty [] = True
isEmpty (x:xs) = False

listLength [] = 0
listLength (x:xs) = 1 + listLength xs
```

Lists & Recursion [2/3]

A function that takes a list will often be recursive. Such a function will usually be organized as follows.

- The version that handles the empty list (**`[]`**) will be the base case.
- The version that handles nonempty lists (**`b:bs`**) will be the recursive case. This will do a computation involving the **head** of the list (**`b`**) and make a recursive call with the **tail** (**`bs`**).

```
myFilter p [] = []   -- p for predicate: function
                     --  returning Bool
myFilter p (x:xs) = if (p x) then x:rest
                     else rest where
    rest = myFilter p xs
```

Note the **`if`** … **`then`** … **`else`** construction. We can put line breaks pretty much anywhere we want inside this construction.

Lists & Recursion [3/3]

Sometimes other kinds of recursion are used. Here is a function that does lookup by index in a list (zero-based).

```
lookup 0 (x:xs) = x
lookup n (x:xs) = lookup (n-1) xs
lookup _ [] = error "lookup: index too big or negative"
```

This pattern means *unused parameter*.

Function **`error`** takes a **`String`** and returns any type; that is, it can be used in any context. It does not actually return anything. Instead, it crashes the program, printing a message that includes the given **`String`**.

An alternate error-message function is **`undefined`**, which takes no parameters. It is like **`error`** with a default message.

```
lookup _ [] = undefined  -- Replaces the above line
```