Haskell: Functions
Haskell: Lists

CS 331 Programming Languages

Lecture Slides

Monday, February 24, 2025

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

ggchappell@alaska.edu

Unit Overview
The Haskell Programming Language

Topics

- ✓ PL feature: type system
- ✓ PL category: functional PLs
- ✓ Introduction to Haskell
- Haskell: functions
- Haskell: lists
- Haskell: flow of control
- Haskell: I/O
- Haskell: data

Review

Review
PL Category: Functional PLs

A typical functional programming language has the following characteristics.

- It has first-class functions.
- It offers good support for *higher-order 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.

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

A value is **mutable if it can be changed. Otherwise, it is **immutable**—like `const` values in C++.

***Code has a **side effect** when it makes a change, other than returning a value, that is visible outside the code.

Review
Introduction to Haskell — Characteristics: Syntax, Type System

Haskell is a pure functional PL.

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

Haskell has a sound static type system with sophisticated type inference. So typing is largely implicit; however, we are allowed to write type annotations, if we wish.

```
square :: Integer -> Integer
square n = n*n
```

Optional type annotation. If this annotation were not present, then we could pass fractional values to function `square`.

Haskell's type system is extensible. We can create new types, but we do not use object-oriented constructions to do so.

Introduction to Haskell — Characteristics: Flow, Evaluation

Haskell has no loops! Instead of iteration, Haskell uses recursion.

However, we often do not make recursive calls explicitly. Instead, we use functions that encapsulate recursive execution.

Encapsulated loop-like construction.

```
> map square [1, 3, 5, 8]
[1,9,25,64]
```

By default Haskell does **lazy evaluation**: expressions are not evaluated until they need to be.

Haskell: Functions

Haskell: Functions
Basic Syntax [1/4]

Comments

- Single line. Two dashes to end of line: `--` … (like Lua)
- Multi-line. Begin with brace-dash, end with dash-brace: `{-` … `-}`

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

There are two kinds of identifiers (terminology below is mine):

- **Normal identifiers** begin with a lower-case letter or underscore. These name variables—including functions.
- **Special identifiers** begin with an UPPER-CASE letter. These name modules, types, and constructors. This is *not* merely a convention!

```
myVariable         -- Normal
_my_Function'_33   -- Normal
MyModule           -- Special
```

For code from this topic, see `func.hs`*.*

Haskell: Functions
Basic Syntax [2/4]

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

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

This represents the GHCi prompt.

```
> addem 2 3
5
```

Parentheses may be used around individual arguments, to override precedence & associativity.

```
addem 18 (5*7)
```

Haskell: Functions
Basic Syntax [3/4]

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

Here is a Fibonacci function using the slow method.

Patterns

```
slowfibo 0 = 0
slowfibo 1 = 1
slowfibo n = slowfibo (n-1) + slowfibo (n-2)
```

We need parentheses here, because function application has high precedence. So "`fibo n-1`" would be the same as "`(fibo n) - 1`".

Haskell: Functions
Basic Syntax [4/4]

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
```

We need parentheses here because function application is left-associative. So "`twice factorial n`" would be the same as "`(twice factorial) n`".

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
```

The tutorial mentioned `let … in`, which puts local definitions *before* the main code. The `where` construction puts them *after* it.

Haskell: Functions
Lambda Functions

A **lambda function** is a kind of expression whose value is a function. In other words, it is a function with no name.

> The name comes from the **lambda calculus,** a mathematical system in which an unnamed function is introduced using the Greek letter lambda (λ).

Haskell introduces a lambda function with a backslash (\), since it looks a little like a lambda.

```
square x = x*x
square' = \ x -> x*x  -- Same
```

Lambda function

```
-- Alternate definitions for addem
addem'     = \ a b -> a+b
addem''    = \ a -> \ b -> a+b
addem''' a = \ b -> a+b
```

Haskell: Functions
Defining Operators [1/3]

We can also define new infix binary operators. As with functions, we write what looks like a call to the operator, an equals sign, and then an expression.

Here is the definition of an operator (`+$+`).

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

Operator names can use any of the following 20 characters. They must not begin with a colon (`:`); those are **special names**.

```
! # $ % & * + . / < = > ? @ \ ^ | - ~ :
```

Haskell: Functions
Defining Operators [2/3]

We can *optionally* set an infix binary operator's precedence and associativity. (Default: precedence 9, left-associative.)

```
a +$+ b = 2*a + b
infixl 6 +$+  -- Left-associative, precedence level 6
              -- Use infixr for right associativity
```

Operators	Precedence	Associativity
Function application	10	Left
`* /`	7	Left
`+ -`	6	Left
`: ++`	5	Right
`== /= < <= > >=`	4	None
`&&`	3	Right
`||`	2	Right

`slowfibo 6`

Haskell **function application** is an invisible operator.

`++` is list concatentation.

`/=` is inequality.

Haskell: Functions
Defining Operators [3/3]

We can use a regular function as an infix binary operator by surrounding its name with backquotes (`).

```
2 `addem` 3
```

And we can use an infix binary operator as a regular function by placing its name in parentheses.

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

Haskell: Functions
Currying [1/2]

Think of a function as *eating* its arguments.

To simulate a multiple-argument function, write a function that eats *one* argument, then returns a function that eats the rest.

Except that second function really only eats *one* (the *second* original argument) and returns a function that eats the rest. Etc.

The last function eats the last argument. Its return value is the final return value—which might not be a function.

This is **currying**, after Haskell Curry: simulating a multiple-argument function using a single- argument function that returns a function.

For an example of currying in C++, see `curry.cpp`*.*

Haskell: Functions
Currying [2/2]

Again, Haskell function application is an invisible operator. It is high-precedence and left-associative [`f a b` is `(f a) b`].

So multiple-argument functions in Haskell are curried. (Right?)

For example, our function `addem` really takes one argument. It returns a function that adds that argument to something. The following are the same, because of left associativity:

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

The intermediate function is not merely theoretical. For example, we can store it in a variable.

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

I imagine that the existence of currying is the reason the Haskell PL was not named "Curry"; the term was already used.

Haskell: Lists

Haskell: Lists
Lists & Tuples [1/5]

A statically typed programming language will typically support two categories of collections of multiple items:

- Collections containing a varying number of items, all of the same type. Examples: C++ `vector`, `list`, `deque`; Java `ArrayList`, `ArrayDeque`.
- Collections containing a fixed number of items, possibly of different types. Examples: C++ `tuple`, `struct`; Java tuple types.

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

For code from this topic, see `list.hs`*.*

Lists & Tuples [2/5]

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
```

Thanks to lazy evaluation, Haskell lists can be infinite.

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

Lists & Tuples [3/5]

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

```
> :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 [4/5]

When looking at lists—particularly infinite lists—a useful function is `take`. This takes a nonnegative integer *count* and a list. It returns a list containing the first *count* items of the given list.

```
> [1, 3 ..]
[1,3,5,7,9,11,13,15,17,19,21,23, … goes on and on …
> take 6 [1, 3 ..]
[1,3,5,7,9,11]
```

If the given list has fewer than *count* items, then the same list is returned.

```
> take 100 [5, 4, 3]
[5,4,3]
```

Lists & Tuples [5/5]

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.

Haskell: Lists
List Primitives [1/2]

A **primitive** (operation) is a fundamental operation that other operations are constructed from.

This terminology is not specific to Haskell.

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 the same; ":" is right-associative
```

Continued …

List Primitives [2/2]

Three Haskell list primitives, continued

3. Pattern matching for lists.

Parentheses are required due to the high precedence of function application.

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

A common convention: read "`x:xs`" as "`x` and some `xs` (plural)".

Pattern matching can be done using `[…, …]` as well.

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

Other List Syntax — Strings

A Haskell `String` is a list of characters (`Char` values). A `Char` literal uses single quotes.

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

The tutorial presented `++` as the `String` concatenation operator. It is, but, more generally, it is the *list* concatenation operator.

```
> "abc" ++ "def"
"abcdef"
> [1,2,3] ++ [4,5,6]
[1,2,3,4,5,6]
```

Haskell: Lists
Other List Syntax — Ranges

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 overloaded functions `enumFromTo`, `enumFromThenTo`, `enumFrom`, and `enumFromThen`, respectively.

```
[1,3..10]
enumFromThenTo 1 3 10  -- Same as above
```

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.

Other List Syntax — List Comprehensions [2/2]

A list comprehension consists of brackets enclosing the following:

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

Examples

Similar to
nested for-loops.

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

Lists & Recursion [1/4]

In a function that takes a list, it is common to have two cases:

- The empty list: `[]`.
- Nonempty lists. A pattern like `b:bs` matches any nonempty list.

TO DO

- Write a function `isEmpty` that determines whether a list is empty.

Done. See `list.hs`*.*

A function taking a list will often be recursive, with the base case handling an empty list, while the recursive case handles a nonempty list. This typically does a computation with the **head** (`b` above) and makes a recursive call on the **tail** (`bs` above).

TO DO

- Write a function `listLength` that returns the length of a list.

Done. See `list.hs`*.*

Prelude function `map` applies a function to each item of a list.

```
> square x = x*x
> square 5
25
> map square [2,5,10,7,1]
[4,25,100,49,1]
```

`map` can always be replaced by a list comprehension. See the first example two slides back.

```
> squareList = map square
> squareList [2,5,10,7,1]
[4,25,100,49,1]
```

Because of currying, we can also think of `map` as a higher-order function. It takes a function that is given a single item and returns a function that is given a list.

TO DO

- Write a function `myMap` that does the same thing as `map`.

Done. See `list.hs`.

Haskell: Lists
Lists & Recursion [3/4]

Prelude function `filter` takes a *predicate* and a list. It returns a list of items that pass the test.

> A **predicate** is a function that returns a Boolean. We can think of it as performing a pass/fail test. (This terminology is not specific to Haskell.)

```
> isBig x = x >= 6
> isBig 5
False
> filter isBig [2,5,10,1,7]
[10,7]
```

> `filter` can always be replaced by a list comprehension. See the third example three slides back.

TO DO

- Write a function `myFilter` that does the same thing as `filter`.

> *Done. See* `list.hs`*.*

Useful: `if` *COND* `then` *EXPR1* `else` *EXPR2*.

- *COND* is an expression of type `Bool`. If it is `True`, then *EXPR1* is returned. If it is `False`, then *EXPR2* is returned.
- Expressions *EXPR1* and *EXPR2* must have the same type.

Lists & Recursion [4/4]

Sometimes other kinds of recursion are used.

TO DO

- Write a function `lookInd` that does item lookup by index (zero-based) in a list.

Done. See `list.hs`*.*

Useful

- The pattern "_" matches any single entity but cannot be used in the right-hand side of a definition. So it means *unused value*.
- `error` is an overloaded function that takes a `String` and returns any type at all. When it executes, it crashes, printing an error message, which will include the given `String`.
- `undefined` is like `error`, but it takes no arguments.