Haskell, Part 1

CS 440: Programming Languages and Translators, Spring 2020

1/16 p.4

Haskell [see LYaH Ch.1]*

- Haskell is a programming language named after the mathematician / logician Haskell Curry.
  - Historical info: You don't need to know this. Curry worked on ”combinatory logic” systems, where you don't define functions by using parameter variables (as in f(x) = x), you define them only by applying and composing smaller functions.
  - For example, there's a basic function I (the identity function). The basic function K takes two arguments, ignores the second, and always returns the first. By starting with an appropriate set of basic functions, you can build any function you can define using parameter variables (though not always in obvious ways).

Back to Haskell the programming language. Its important features:

- **Pure**, **functional**, **lazy** language with a **rich static** type system.
- **Pure** - there are no side effects in functions, expressions
  - Evaluation of an expression within a fixed environment always returns the same result.
  - Once you give a variable x a value, it has that value as long as it's alive.
  - No assignment operator, +=, ++, etc.
  - Promotes **referential transparency** (you can substitute equals for equals; see more below)
  - I/O is an exception – inherently impure operation
- **Functional** – includes higher-order functions, functions as "1st class" citizens
  - **Higher-order** functions - functions as parameters, functions as results
    - Operations on functions - composition, ...
    - E.g., integral of f(x) dx from a to b --- integral is a higher-order function
    - Generally use recursion instead of iteration
  - **First-class functions** – You can use functions as expressions just like any other kind of expression. (Can have variables whose values are functions, build pairs and arrays of functions, etc.)
- **Lazy** - Don't do an operation unless you need the result. (Cf. **eager** – you evaluate something before deciding you need it.)
  - In C, function arguments are eager: Define ite(int x, int y, int z) { if (x) return y; else return z; }. Here, if e_2 causes an error and e_0 is false, then ite(e_0, e_1, e_2); will cause an error.
  - The lazy parts of C are the conditionals. E.g., (e_0 ? e_1 : e_2) executes either the true branch or the false branch but not both. If e_0 is true, it doesn't matter whether evaluating e_2 would cause an error or not.
  - Lazy evaluation can speed things up (you don't evaluate something you wind up not needing).

* *LYaH* = *Learn You a Haskell For Great Good!* by Miran Lipovača, http://learnyouahaskell.com/

- Can lazy evaluation slow things down?
  - What if we want to repeatedly evaluate an expression?).
  - We could evaluate it once and cache the result, but that only works if the second, third, etc. evaluations would return the same value.
  - (This is why purity is useful.)
- Lazy evaluation makes figuring out resource use harder, generally speaking.

- **Rich type system [not in LYaH]**
  - **Static** typing / typechecking: Types can be verified for correctness at "compile" time.
  - **Recursive user-defined types allowed** – Haskell allows definition of complicated datatypes like trees – this makes types and typechecking more complicated.
  - **Type inferencing** – In general, Haskell compiler can figure out what types your identifiers and expressions must have. This reduces burden of having complicated types.
  - User-defined **polymorphic** types – You can write functions / expressions that have multiple types simultaneously.
    - **Type classes** - generalizes notion of operator overloading.
    - **Type parameters** – you can say something is of type *t* where *t* ranges over a set of types.
    - We'll see more about these two later.

Referential Transparency; Purity; Confluence [not in LYaH]

- An expression is **referentially transparent** if in any context, it can always be replaced by its value (or its value can be replaced by the expression). A language is referentially transparent if all its constructs are. (The opposite of referential transparency is **referential opacity**.)
- Say expression e is referentially transparent and its value equals the constant c. Then if e occurs in any larger expression, we can replace it by c; if c occurs in an expression, we can replace it by e.
- For example, we can transform c == c into e == c. Since constants don't change value, for e to be referentially transparent, it must *always* evaluate to c. If e is **pure** (its evaluation has no side effects), then e is referentially transparent; if e has side effects, it's not pure and not referentially transparent. (So programming languages like C and Java are not referentially transparent.)
- Purity and referential transparency may seem strange, but it's actually how we do algebra — when we define $p(x) = x^2 + x$ and then argue that $p(5)/2 = (5^2 + 5)/2$, we're using referential transparency.
- If an expression is referentially transparent, then we can rewrite its subexpressions in any order: It doesn't matter what order we expand p(5) + p(6); we always get the same **unique result**. Since order of evaluation doesn't matter, algebra is said to be confluent. Referential transparency implies confluency.
- Turning to Haskell, since Haskell expressions are pure, Haskell supports referential transparency and is confluent. We can evaluate Haskell programs by replacing expressions and subexpressions by their value until we come up with the unique result at the end.

- Programming languages in general have a referentially transparent subset of operations (think arithmetic expressions in C, for example). In Haskell, all its operations are referentially transparent.
- Two points we'll put off for a while. First, I/O is inherently **impure** (reading in a value for x twice can give you different values), so it takes some work to make Haskell support I/O. Second, Haskell's lazy evaluation is a subtle part of why it's completely referentially transparent.
- Pure functional languages are often thought of as being more concise and "elegant", basically because they combine first-class functions (which make it easier to define complex actions) and confluence (which makes it easier to explain program execution). [Or at least, that's my view.]

Running Haskell [LYaH Ch.2]

- Haskell platform - **Glasgow Haskell Compiler** (GHC) - see `http://www.haskell.org/ghc`
- Runs on lots of platforms, pretty easy to install on Mac OS and Linux (don't know about Windows). I'll let you investigate how to get a copy of it running on your laptop. We'll get it running on fusion1.cs.iit.edu.
- An interactive version lets you do read / eval / print loops.

Sample Interactive Run of Haskell

```
unix > ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help
Prelude> 3 + 5  -- typical read/eval/print loop
8
Prelude> :set prompt "> " -- changes the prompt
> it     -- value of previous eval
8
> 5 + it -- new version of it is 5 + (old version of it)
13
> it
13
```

- Arithmetic: `+`, `-`, `*`, `/`, `mod` (not `%`), `rem`. Unary - is an operator that doesn't bind very strongly

```
> 2 * (-4)
8
> 2 * -4
<interactive>:77:1: error:
    Precedence parsing error …
```

- Booleans: type `Bool`, values `True`, `False`, `&&`, `||`, `not` [not !]

```
> 2 == 1+1 && 4 /= 5 || 0 > 2 -- note inequality is /=
True
> if 2 > 3 then 1 else 5 -- conditional expr: (2 > 3 ? 1 : 5) in C
5
```

Simple Functions [LYaH Ch.2, p.4]

```
> sqrt 4 -- function application doesn't require parens
2.0
> sqrt(4) + 4   -- redundant parens are allowed
6.0
> sqrt 4 + 4    -- + has lower precedence than fcn application
6.0
> sqrt (4 + 4)  -- parens required
2.8284271247461903
> f(x) = x + 2 -- function definition
> f x = x + 2  -- drop redundant parens
> f 3
5
> f (f 3) -- need parens bec. function application is left associative [1/16]
7
> g x y = x * y + y + f(x * y) -- function of two arguments
> g 2 3   -- call function with two arguments
17
> -- can say things like
> h = g 2 -- like saying h y = 2 * y + y + f(2 * y)
> h 3
17
```

Basic Lists [LYaH Ch.2, p.6]]

- Lists use square bracket notation. Empty list is `[]`. Use `:` for cons (adding an element to a list), `++` for concatenation

```
> []
[]
> 1 : [2,3,4] -- prepend ("cons") value
[1,2,3,4]
> 1 : 2 : 3 : 4 : [] == [1,2,3,4]
True
> [1,2,3] ++ [4,5] -- list concatenation
[1,2,3,4,5]
```

- All the list elements must be of the same type

```
> ['a', 17]
<interactive>:128:7: error:
   …
```

- Functions `head` and `last` return the first and last element respectively. Function `tail` returns all but the head; `init` returns all but the last element.

```
> [1,2,3,4] -- List of 4 elements
```

```
[1,2,3,4]
> head [1,2,3,4]
1
> tail [1,2,3,4]
[2,3,4]
> init x       -- all but last element
[1,2,3,5]
> last x       -- last element
7
```

- Taking `head`, `tail`, `init`, or `last` of `[]` yields a runtime error

```
> tail []
*** Exception: Prelude.tail: empty list
```

- It's ok to have a list of lists where the sublists have different lengths

```
> [[1+1], [3,2*3+4]]
[[2],[3,10]]
```

- We can compare lists elementwise. E.g., we can compare lists of integers.

```
> [0,2] < [1] && [1,2] < [1,5] && [3,4] < [3,4,5]
True
```

- We can compare lists of any kind of base value that supports comparison, so lists of (lists of integers) are comparable too.

```
> [1,2] < [1,5]
True
> [3,4] < [3,4,5]
True
> [[1,2],[3,4]] < [[1,5],[3,4,5]] -- [1,2] < [1,5] (so we don't have to
test) [3,4] < [3,4,5]? [1/14]
True
```

- Strings are just lists of characters

```
> "abc" == ['a', 'b', 'c']
True
> "" == []       -- empty string is just empty list of characters
True
> "abc" ++ "def" -- ++ on strings is string concatenation
"abcdef"
> "a" < "ab" && "ab" < "ac" -- string comparisons are list comparisons
True
```

- Exist `length, null, reverse, minimum, maximum, product` functions

```
> null [1,2]
False
> null []
```

```
    True

-- take n x returns list of first n elements of list x
-- drop n x returns what remains after omitting first n elements of list x
-- If n > length of x, then take n x and drop n x return []
    > x
    [1,2,3,5,7]
    > take 3 x
    [1,2,3]
    > take 4 x
    [1,2,3,5]
    > take 0 x
    []
    > take (-1) x      -- actually do need those parentheses†
    []
    > take 4 []
    []
    > drop 3 x         -- list without first three items
    [5,7]
    > drop 8 x         -- 8 is > length of x
    []
```

List Ranges [LYaH Ch.2, p.11]

```
      > [1..5] -- list range, 1 upward to 5
      [1,2,3,4,5]
      > [1, 3..10] -- list range: delta (3-1) between elements
      [1,3,5,7,9]
      > [10..0]    -- 10 > 0 so we're done immediately
      []
      > [10, 9..0] -- need to be explicit for downward-going lists
      [10,9,8,7,6,5,4,3,2,1,0]
      > [10, 8..0] -- delta -(10-8) between elements
      [10,8,6,4,2,0]
```

List comprehensions [LYaH Ch.2, p.12]

- List comprehensions are similar to set comprehensions

```
      > [x*x | x <- [1..5]] -- list of squares of 1 through 5
      [1,4,9,16,25]
```

† There aren't negative constants in Haskell: `-1` is actually a function call of unary `-` on argument 1. Saying `take -1 x` means `(take -) 1 x`; it tries to run `take` on function `-` instead of an integer.

```
> [ [x*x] | x <- [1..5]] -- list of singleton lists of squares
[[1],[4],[9],[16],[25]]
> [ x*x | x <- [1..100], x*x < 64] -- use filter to get squares of 1 to
100 stopping once a square is >= 64
[1,4,9,16,25,36,49]
> [ x+y | x <- [1..5], y <- [3..9]] -- can choose from multiple lists
[4,5,6,7,8,9,10, -- 1 + each of 3..9
5,6,7,8,9,10,11,   -- 2 + each of 3..9
6,7,8,9,10,11,12,
7,8,9,10,11,12,13,
8,9,10,11,12,13,14]
```

[1/14] `-- Use list comprehensions to run a whole bunch of operations`

```
> [f [1,2,3,5,7] | f <- [length, minimum, maximum, sum, product]]
[5,1,7,18,210]
```

Activity Questions, Lecture 1

(These are pretty simple questions so I probably won't post solutions.)

1. What is purity and makes a language pure?
2. What are higher-order functions? First-class functions?
3. What's the difference between lazy and eager evaluation? Does lazy evaluation make programs faster or slower? (Or maybe both?)
4. What makes typechecking static? Does it make programming easier or harder? (Or both?)
5. What is type inference? Does it make programming easier or harder? (Or both?)
6. What is referential transparency? What is its connection to purity?
7. Why is confluence? Why is it helpful? How does referential transparency figure in?
8. All of Haskell is referentially transparent. Do C & Java have any referentially transparent parts?
9. What's the Haskell syntax for function application? How does it differ from, say, C's or Java's?
10. What do you get if you remove all the redundant parentheses from (sqrt(x) - y) * sqrt(u+v) ?
11. What, if any, is the difference in values of [1,2,3] and 1 : 2 : 3 : [] and [1,2] ++ [3] ?
12. What is the easier way to write ['a', 'b', 'c']?
13. What is head "LMN"? tail "LMN"?
14. What is a list range? What is the list range syntax for 0, 1, …, 6? How about 6, 5, 4, …, 0?
15. What is list comprehension? Give a list comprehension for $1^3, 2^3, 3^3, \ldots, 10^3$.
16. What does [x | x <- [1..10], x*x > 24] evaluate to? Describe this expression in English.

(No solutions)