

CS421 Lecture 7: Hask¹

Mark Hills

`mhills@cs.uiuc.edu`

University of Illinois at Urbana-Champaign

June 16, 2008

¹Based on slides by Mattox Beckman, as updated by Vikram Adve, Gul Agha, and Elsa Gunter

Putting Concepts to Practice

At this point, we have a good idea of what syntax is, and of what *lexemes*, or *tokens*, in the language are. We now want to put this to use in taking a programming language and figuring out which tokens are used in any given program – or figuring out that there is a lexical error in the source code. Our language will be

Putting Concepts to Practice

At this point, we have a good idea of what syntax is, and of what *lexemes*, or *tokens*, in the language are. We now want to put this to use in taking a programming language and figuring out which tokens are used in any given program – or figuring out that there is a lexical error in the source code. Our language will be

Hask

(a shorter Haskell...)

Hask

Hask is a (partially) lazy functional language that is a subset of the language Haskell. The concepts are similar to what we have looked at so far for OCaml, with some significant differences that should make it interesting.

Simple Expressions

A variety of simple expressions are available in the language. This includes arithmetic expressions, relational expressions, and logical expressions.

Arithmetic Expressions

```
1 1 + 2 ;; -- 3
2 3 - 1 ;; -- 2
3 4 * 3 ;; -- 12
4 4 / 2 ;; -- 2
5 4 ** 2 ;; -- 16
```

Logical Expressions

```
1 1 < 2 ;; -- True
2 1 > 2 ;; -- False
3 1 == 2 ;; -- False
4 1 /= 2 ;; -- True
```

Boolean Expressions

```
1 True && False ;; -- False
2 True || False ;; -- True
3 not True ;; -- False
```

Variable Bindings

Top-level variable bindings do not require a `let` (although we have one, which we'll see in a bit):

```
1 x = 5 ;;  
2 y = 6 ;;  
3 x + y ;; -- 11
```

Functions

Functions are the core model of computation in Haskell, much like in OCaml. One difference is that we will treat function parameters *lazily* – instead of evaluating them first (often called *eager* or *strict* evaluation), we will wait to evaluate them until we find a use of them in the function body.

Functions

```
1 [- Sum two numbers -]
2 fun sum x y = x + y ;;
3
4 [- Double a given number -]
5 fun double x = 2 * x ;;
6
7 [- Composition -- double sum result -]
8 fun sum_then_double = sum . double ;;
```

Anonymous Functions

```
1 [- Anonymous version of sum -]
2 (\ x y -> x + y) 2 3 ;; -- 5
3
4 [- Application function -]
5 (\ f x -> f x) (\ x -> x + 1) 3 ;; -- 4
```

Type Expressions

We can also use type expressions to say what the expected type of a function is. We have type inference, but this provides good documentation of our intentions.

```
1  [- Sum two numbers -]
2  sum :: Int -> Int -> Int ;;
3  fun sum x y = x + y ;;
4
5  [- Double a given number -]
6  double :: Int -> Int ;;
7  fun double x = 2 * x ;;
8
9  [- Composition -- double sum result -]
10 sum_then_double :: Int -> Int -> Int ;;
11 fun sum_then_double = sum . double ;;
```

Let Bindings

Local bindings come in two forms. The first is the `let`, which is familiar from OCaml, although we always have an `in` portion of the `let`.

```
1 [- Single binding -]
2 let x = 5 in x ** x ;; -- 3125
3
4 [- More complex, with multiple bindings -]
5 let f = (\ x -> x + x) ; g = 3 in f g ;; -- 6
```

Where Bindings

The second form of binding is the `where` binding, which provides a method of giving local bindings in a function. The main difference is that the `where` portion comes *after* the function body. Note that `and` and `end` aren't used in Haskell, but are here to make our life easier.

```
1 fun sum_double x y = double (sum x y)
2   where fun sum a b = a + b
3         and fun double a = 2 * a
4         end
5 ;;
6
7 sum_double 2 3 ;; -- 10
```

Conditionals

We also have a conditional expression, which always must have both branches.

```
1 if True then 5 else 6 ;; -- 5
2
3 (if False then (\ x -> x + 1)
4   else (\ x -> x + 2)) 10 ;; -- 12
```

Lists

List have similar syntax to OCaml, but with some slight differences.

```
1 [1,2,3] ;; -- [1,2,3]
2
3 1 : [2,3] ;; -- [1,2,3]
4
5 [1,2] ++ [3] ;; -- [1,2,3]
6
7 [1 .. 10] ;; -- [1,2,3,4,5,6,7,8,9,10]
```

The latter is a *list comprehension* – we will only use the simple form shown here, although much more complex forms are allowed.

Tuples

Tuples are also similar to those in OCaml, using the same syntax. We will restrict our use to pairs, since this lets us have predefined operations to get the values back out.

```
1 (1,2) ;; -- pair of integers
2
3 fst (1,2) ;; -- 1
4
5 snd (1,2) ;; -- 2
6
7 snd ((\ x -> x), (\ y -> y)) ;; -- \ y -> y
```

Comments

You've probably noticed two comment forms used in the above examples.

- `--` is a line comment, and will comment out anything through the end of the line
- `[--` begins a block comment, with `-]` then ending the comment; block comments can be nested

Picky Details

- All characters will be from the ASCII character set
- Lowercase letters are all letters from a to z
- Uppercase letters are all letters from A to Z
- Digits are all numbers from 0 to 9
- Whitespace includes spaces, tabs, and newlines
- Identifiers for data items (variables, functions) all start with a lowercase letter, and then consist of 0 or more lowercase letters, uppercase letters, digits, underscores, and single quotes
- Identifiers for types are defined identically to those for data items except they start with an uppercase letter

Picky Details, cont.

- Numbers are all integers, made up of one or more digits, potentially with a leading sign
- Strings are surrounded with double quotes and can include any character typeable on a standard keyboard, plus standard escape characters, indicated with a preceding backslash
- Parens can be used to surround parts of expressions, and impact precedence (which we will hit in parsing)
- The empty parens, `()`, represent the Unit type, like in OCaml