HMC CS 131

Week 1, Lecture 1: Introduction to Haskell and Functional Programming

Today's Topics

- The Read-Eval-Print Loop (REPL)
- A Tour of Haskell:
  - Basic values and types (numerics, booleans, characters, strings, lists, tuples)
  - Calling functions
  - Variables, bindings, and scope
  - Writing code in files
  - Defining functions
  - Types
- Functional Programming: definitions and foundations
- Abstraction: The Essence of CS 131

Haskell: Getting Started

REPL The REPL (Read-Eval-Print Loop) is an interactive environment for executing Haskell code. It consists of the following steps:

1. **Read**: Read the input from the user
2. **Eval**: Evaluate the input code
3. **Print**: Print the result of the evaluation
4. **Loop**: Go back to the Read step

REPL Example: GHCI To use the REPL in Haskell, you can launch `ghci` (Glasgow Haskell Compiler Interactive) in your terminal.

Haskell: Basic Values and Types

Haskell has a variety of basic values and types, as shown in the table below:

Value	Type	Description
1, 2, 100000000000, -42, ...	Integer	Whole numbers
3.14, 3.2831, -2.718, ...	Double	Floating-point numbers
True, False	Bool	Booleans
'a', 'z', ...	Char	Characters
"hello", "world"	String	Strings
[1, 2, 3, 4]	[Integer]	List of Integers
(True, 0.5)	(Bool, Double)	Tuple of Bool and Double

Note: All type names are upper-case! However, not all upper-case names are types.

Operations Haskell has several built-in operations, including:

- Arithmetic: `+`, `-`, `*`, `/`
- Comparison: `==`, `/=`, `>`, `<`, `>=`, `<=`
- Logical: `&&`, `||`
- Concatenation: `++`

Haskell: Calling Functions

Haskell is a functional programming language, and calling functions is straightforward. Here are some examples:

```
sqrt 4
show 42
min 0 42
max 0 42
-- Incorrect: max 100 42 + 1
-- Correct: max 100 (42 + 1)
-- Incorrect: max -42 42
-- Correct: max (-42) 0

not True
4 / 2   -- vs   div 4 2
div 2 4
mod 2 4
mod 4 2
mod 4 3
mod (10 + 7) 12
```

Watch out! `/` is floating-point division, while `div` is integer division.

Watch out! Enclose subexpressions in parentheses.

Haskell: Variables, Bindings, and Scope

Some Vocabulary

- **Binding** (noun) and **bind** (verb): Associate a variable name with an expression. Similar to assignment (noun) and assign (verb).
- **Value** (noun): The expression to which a name is bound.
- **Reference** (verb): Use a name in an expression. During evaluation, the name is just an alias for the expression.
- **Scope**: The region of the program where a name can be referenced. A variable's scope depends in part on where the variable is bound.
- **Lookup** (verb) or **resolve** (verb): Evaluate a name; find its value.

All variable names are lower-case!

```
x = 21
y = 2 * x
```

Top-level bindings (think "global variables"): The scope of a top-level binding is the entire file (or the rest of the `ghci` session).

Trick question: What's the value of `x`?

```
x = 24
x = 41
x = x + 1
```

If we type this code into a file and try to run it, we will get an error. In `ghci`, we can type the first and second lines, and get the value of `x`. We can type the third line, but when we try to get the value of `x`, `ghci` goes into an infinite loop!

A scope can bind a name at most once. In other words, all "variables" are constants.

Local bindings (think "local variables"):

```
let x = 21 in x + x
```

The scope of a `let` variable is the body of the `let`. The result of the `let` expression is the result of evaluating the `let` body.

`let` and `in` are keywords in Haskell.

A scope can bind a name at most once. In other words, all "variables" are constants.

Scopes can nest:

```
let x = 21 in let x = 10000 in x
```

Haskell resolves names from inner-most to outer-most scope.

Haskell: Writing Code in Files

Plus: comments, loading & reloading code in `ghci`.

Haskell code files have a `.hs` extension. To add comments to your code, use the following syntax:

```
-- Single-line comment
{-
   Multiline comment
-}
```

Top-level bindings can go in a file, but expressions on their own can't go in a file. Don't click the Run button on repl.it. Load the code into `ghci` instead.

If you have a `<file>.hs` file, you can load it into `ghci` with:

`:load <file>`

or

`:l <file>`

If you have already loaded a file, then you modify the file and save it again, you can reload the contents of the file:

`:reload`

or

`:r`

Haskell: Defining Functions

Plus: local definitions and pattern matching.

Haskell is a functional programming language, and functions are easy to define.

Function Calls

```
successor 1
average 0 10
```

Function Definitions

```
successor n = n + 1
average x y = (x + y) / 2
```

The function's body is an expression. The result of calling the function is the result of evaluating the function's body, with the argument values bound to the corresponding parameter names.

Local Bindings in a Function Three different ways to define x^2 + 1/x^2:

```
f x = x * x + 1 / (x * x)

f x = let value = x * x
      in value + 1 / value

f x = value + 1 / value
  where value = x * x
```

`let` expressions are one way to do local variables in a Haskell function. As always, the scope of the `let` binding is the body of the `let` expression.

`where` clauses are another way to do local variables in a Haskell function. The scope of the `where` clause is the body of the function (including the `where` clause).

Which of these definitions you choose to write is largely a matter of style or convenience.

Pattern Matching Pattern matching is one of Haskell's superpowers.

A typical definition of `fact` that uses `if`/`then`/`else`:

```
fact n = if n == 0
         then 1
         else n * fact (n - 1)
```

A definition of `fact` that uses pattern matching:

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

When a program calls `fact`, Haskell tries to match the call's argument against each pattern, from top to bottom. As soon as it finds a match, Haskell uses that version of the function.

Pattern Matching in Haskell Pattern matching is one of Haskell's most powerful features, allowing you to write elegant, concise, and readable code. It is a mechanism that enables you to destructure data structures and execute different code based on the shape and content of the data. In this section, we'll dive into the details of pattern matching in Haskell and see how it can make your code more expressive.

In Haskell, pattern matching can be used in various contexts such as function definitions, case expressions, and list comprehensions. We'll start by exploring pattern matching in function definitions.

Imagine we want to implement a factorial function, `fact`. A typical approach using `if`/`then`/`else` would look like this:

```
fact n = if n == 0
         then 1
         else n * fact (n - 1)
```

However, we can rewrite this function more elegantly using pattern matching:

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

In this version of the `fact` function, we define two separate equations for the function. The first equation has the pattern `0` as its argument, and the second one has the variable `n`. When a program calls `fact`, Haskell tries to match the call's argument against each pattern, from top to bottom. As soon as it finds a match, Haskell uses that version of the function. If the argument is `0`, the first equation will match, and the function will return `1`. If the argument is any other number, the second equation will match, and the function will continue with the recursive call.

Pattern matching can also be used with more complex data structures like lists and custom data types. For example, let's implement a function `head'` to get the first element of a list:

```
head' :: [a] -> a
head' [] = error "Empty list"
head' (x:_) = x
```

Here, we use two patterns: an empty list `[]` and a non-empty list `(x:_)`. The first pattern matches an empty list and raises an error. The second pattern matches a list with at least one element, represented by `x`, followed by an underscore (`_`). The underscore is a wildcard pattern, meaning it will match anything, but we won't use its value in the function body. In this case, the function returns the first element `x`.

Pattern matching can also be used in case expressions. For instance, let's rewrite the `head'` function using a case expression:

```
head' :: [a] -> a
head' xs = case xs of
             [] -> error "Empty list"
             (x:_) -> x
```

This version of `head'` behaves the same as the previous one, but it demonstrates how you can use pattern matching in a case expression.

In conclusion, pattern matching is a powerful and versatile feature in Haskell that enables you to write clean and expressive code. By using pattern matching, you can easily destructure data structures, execute different code based on input patterns, and improve code readability.

Types in Haskell

Some Vocabulary

- **type-safe** (adjective): a program that has no type errors
- **type-check** (verb): determine whether a program has type errors
- **static** (adjective): a property of a program that is (or can be) determined without running the program
- **dynamic** (adjective): a property of a program that is (or can be) determined by running the program
- **statically typed** (adjective): a programming language with static type checking. Typically, a compiler performs type checking. If the compiler finds a type error, the program does not compile (and therefore can never run).
- **dynamically typed** (adjective): a programming language with dynamic type checking. An interpreter might perform type-checking as the program runs. If the program has a type error, the program may crash.

- **type annotation** (noun): information that the programmer provides about the type of a variable, function, etc.
- **type inference** (noun): a form of type checking that does not require (a lot of) type annotations

Haskell is a **statically typed language** that uses **type inference**.

Examining Types in `ghci` We can ask `ghci` for the type of an expression using:

```
:type <expr>
```

or

```
:t <expr>
```

Haskell is a statically typed language that uses type inference. We are usually not required to write type annotations, but it's still a good idea.

Function Types in Haskell

```
successor n = n + 1
average x y = (x + y) / 2
```

Function Types in Haskell

```
successor :: Integer -> Integer
successor n = n + 1

average :: Double -> Double -> Double
average x y = (x + y) / 2
```

"successor is a function that takes an Integer and results in an Integer."

"average is a function that takes two Doubles and results in a Double."

Functional Programming

Some Basic Concepts What is functional programming?

In functional programming, computation proceeds by evaluating expressions. This is in contrast to imperative programming, where computation proceeds by updating state. In pure functional programming, evaluating expressions is the only way computation proceeds. Haskell is a pure, functional programming language.

The Essence of CS 131

Abstraction "Refactoring": Eliminating Repeated Code

Consider the following expressions:

```
2 * 2 + 1
...
3 * 2 + 1
...
... (4 * 2 + 1) ..
...
5 * 2 + 1
```

We can "factor out" the operation into a helper function:

```
f :: Integer -> Integer
f n = n * 2 + 1
```

Now, we can replace the repeated code with calls to the helper function:

```
f 2
...
f 3
...
... (f 4) ..
...
f 5
```

"Refactoring": Step-by-step

1. Identify the repeated code:

   ```
   2 * 2 + 1
   ...
   3 * 2 + 1
   ...
   ... (4 * 2 + 1) ..
   ...
   5 * 2 + 1
   ```

2. Write a function with the appropriate type signature:

   ```
   f :: Integer -> Integer
   f n = n * 2 + 1
   ```

3. Replace the repeated code with calls to the helper function:

   ```
   f 2
   ...
   f 3
   ...
   ... (f 4) ..
   ...
   f 5
   ```

Functions are parameterized expressions.

Abstraction is the essence of CS 131.