What is a Functional Language

Hard to give a precise definition, but generally speaking:

- Functional programming is a style of programming in which the basic method of computation is functions application.
- A functional language is one that supports and encourages the functional style.

However, higher-order functions and the possibility to treat functions as data are commonly accepted criteria.

Example (1)

Summing the integers from 1 to 10 in Java:

```java
total = 0;
for (i = 1; i <= 10; ++i)
    total = total + 1;
```

The method of computation is to execute operations in sequence, in particular variable assignment.

Example (2)

Summing the integers from 1 to 10 in Haskell:

```haskell
sum [1..10]
```

The method of computation is function application.

Of course, essentially the same program could be written in Java, but:

- it would be far more verbose
- for most purposes, it wouldn't be a “good” Java program: this is simply not how one programs in Java.
This Lecture

- First steps
- Types in Haskell
- Defining functions
- Recursive functions
- Declaring types

The GHC System (1)

- GHC supports Haskell 98 and many extensions
- GHC is currently the most advanced Haskell system available
- GHC is a compiler, but can also be used interactively: ideal for serious development as well as teaching and prototyping purposes

The GHC System (2)

On a Unix system, GHCi can be started from the ghci:

```
isis-1% ghci
```

```
/ _ \\ / _ \GHCI Interactive, version 6.3, for Haskell 98.
/ / _\ / /_|_ http://www.haskell.org/ghc/
/ / _\ / _ \ _
\ _ _ / /_/ Type :? for help.

Loading package base ... linking ... done.
Prelude>
```

The GHC System (3)

The GHCi > prompt means that the GHCi system is ready to evaluate an expression. For example:

```
> 2+3*4
14
```

```
> reverse [1,2,3]
[3,2,1]
```

```
> take 3 [1,2,3,4,5]
[1,2,3]
```
Function Application (1)

In mathematics, function application is denoted using parentheses, and multiplication is often denoted using juxtaposition or space.

\[ f(a, b) + c \cdot d \]

“Apply the function \( f \) to \( a \) and \( b \), and add the result to the product of \( c \) and \( d \).”

Function Application (2)

In Haskell, *function application* is denoted using *space*, and multiplication is denoted using *\(*\).*

\[ f \ a \ b + c \* d \]

Meaning as before, but Haskell syntax.

Function Application (3)

Moreover, function application is assumed to have *higher priority* than all other operators. For example:

\[ f \ a + b \]

means

\[ (f \ a) + b \]

not

\[ f (a + b) \]

What is a Type?

A *type* is a name for a collection of related values. For example, in Haskell the basic type

\[ \text{Bool} \]

contains the two logical values

\[ \text{False} \]
\[ \text{True} \]
Types in Haskell

- If evaluating an expression \( e \) would produce a value of type \( t \), then \( e \) has type \( t \), written \( e :: t \).
- Every well-formed expression has a type, which can be automatically calculated at compile time using a process called *type inference* or *type reconstruction*.
- However, giving manifest type declarations for at least top-level definitions is good practice.

Basic Types

Haskell has a number of *basic types*, including:

- **Bool**: Logical values
- **Char**: Single characters
- **String**: Strings of characters
- **Int**: Fixed-precision integers

List Types

A list is a sequence of values of the *same* type:

\[
[\text{False}, \text{True}, \text{False}] :: [\text{Bool}]
\]

\[
[\text{'a'}, \text{'b'}, \text{'c'}, \text{'d'}] :: [\text{Char}]
\]

In general:

\([t] \) is the type of lists with elements of type \( t \).

Tuple Types

A tuple is a sequence of values of *different* types:

\[
(\text{False}, \text{True}) :: (\text{Bool}, \text{Bool})
\]

\[
(\text{False}, \text{'a'}, \text{True}) :: (\text{Bool}, \text{Char}, \text{Bool})
\]

In general:

\((t_1, t_2, \ldots, t_n)\) is the type of \( n \)-tuples whose \( i^{th} \) component has type \( t_i \) for \( i \in [1 \ldots n] \).
Function Types (1)

A function is a mapping from values of one type to values of another type:

\[
\text{not} :: \text{Bool} \rightarrow \text{Bool}
\]

In general:

\[
t_1 \rightarrow t_2\]
is the type of functions that map values of type \(t_1\) to values to type \(t_2\).

Function Types (2)

If a function needs more than one argument, pass a tuple, or use currying:

\[
(\&\&) :: \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool}
\]

This really means:

\[
(\&\&) :: \text{Bool} \rightarrow (\text{Bool} \rightarrow \text{Bool})
\]

Idea: arguments are applied one by one. This allows partial application.

Polymorphic Functions (1)

A function is called polymorphic (“of many forms”) if its type contains one or more type variables.

\[
\text{length} :: [a] \rightarrow \text{Int}
\]

“For any type \(a\), \text{length} takes a list of values of type \(a\) and returns an integer.”

This is called Parametric Polymorphism.

Polymorphic Functions (2)

The type signature of length is really:

\[
\text{length} :: \forall a . [a] \rightarrow \text{Int}
\]

- It is understood that \(a\) is a type variable, and thus it ranges over all possible types.
- Haskell 98 does not allow explicit \(\forall\)s: all type variables are implicitly qualified at the outermost level.
- Haskell extensions allow explicit \(\forall\)s.
Types are Central in Haskell

Types in Haskell play a much more central role than in many other languages. Two reasons:

• Haskell’s type system is very expressive thanks to Parametric Polymorphism:
  
  
  (++): [a] -> [a] -> [a]

• The types say a lot about what functions do because Haskell is a pure language: no side effects (Referential Transparency)

Conditional Expressions

As in most programming languages, functions can be defined using conditional expressions:

\[
abs :: \text{Int} \rightarrow \text{Int} \\
abs n = \text{if } n \geq 0 \text{ then } n \text{ else } -n
\]

Alternatively, such a function can be defined using guards:

\[
abs :: \text{Int} \rightarrow \text{Int} \\
abs n \mid n \geq 0 \Rightarrow n \\
\mid \text{otherwise} \Rightarrow -n
\]

Pattern Matching (1)

Many functions have a particularly clear definition using pattern matching on their arguments:

\[
\text{not} :: \text{Bool} \rightarrow \text{Bool} \\
\text{not False} = \text{True} \\
\text{not True} = \text{False}
\]

Pattern Matching (2)

Case expressions allow pattern matching to be performed wherever an expression is allowed, not just at the top-level of a function definition:

\[
\text{not} :: \text{Bool} \rightarrow \text{Bool} \\
\text{not b = case b of} \\
\text{False } \Rightarrow \text{True} \\
\text{True } \Rightarrow \text{False}
\]
List Patterns (1)

Internally, every non-empty list is constructed by repeated use of an operator (:) called “cons” that adds an element to the start of a list, starting from [], the empty list.

Thus:

\[ [1,2,3,4] \]

means

\[ 1:(2:(3:(4:[]))) \]

List Patterns (2)

Functions on lists can be defined using \( x:xs \) patterns:

- head :: \[a\] -> a
  - head (x:_ \[a\]) = x

- tail :: \[a\] -> \[a\]
  - tail (_:xs \[a\]) = xs

Lambda Expressions

A function can be constructed without giving it a name by using a lambda expression:

\[ \lambda x \rightarrow x + 1 \]

“The nameless function that takes a number \( x \) and returns the result \( x + 1 \)”

Note that the ASCII character \( \backslash \) stands for \( \lambda \) (lambda).

Why Are Lambda’s Useful?

Lambda expressions can be used to give a formal meaning to functions defined using currying.

For example:

\[ \text{add} \ x \ y = x+y \]

means

\[ \text{add} = \lambda x \rightarrow (\lambda y \rightarrow x+y) \]
Recursive Functions (1)

In Haskell, functions can also be defined in terms of themselves. Such functions are called *recursive*. For example:

```haskell
factorial 0 = 1
factorial n | n >= 1 = n * factorial (n - 1)
```

Recursive Functions (2)

Why does this work? Well, consider:

```
factorial 3
= 3 * factorial 2
= 3 * (2 * factorial 1)
= 3 * (2 * (1 * factorial 0))
= 3 * (2 * (1 * 1))
= 3 * (2 * 1)
= 3 * 2
= 6
```

Why Is Recursion Useful?

• Some functions, such as factorial, are *simpler* to define in terms of other functions.
• As we shall see, however, many functions can *naturally* be defined in terms of themselves.
• Properties of functions defined using recursion can be proved using the simple but powerful mathematical technique of *induction*.

Recursion on Lists (1)

Recursion is not restricted to numbers, but can also be used to define functions on lists. For example:

```haskell
product :: [Int] -> Int
product [] = 1
product (n:ns) = n * product ns
```
Recursion on Lists (2)

product [2, 3, 4]
= 2 * product [3, 4]
= 2 * (3 * product [4])
= 2 * (3 * (4 * product []))
= 2 * (3 * (4 * 1))
= 24

Data Declarations (1)

A new type can be declared by specifying its set of values using a data declaration. For example, Bool is in principle defined as:

data Bool = False | True

Data Declarations (2)

What happens is:

- A new type Bool is introduced
- Constructors (functions to build values of the type) are introduced:
  
  False :: Bool
  True :: Bool
  
  (In this case, just constants.)
- Since constructor functions are bijective, and thus in particular injective, pattern matching can be used to take apart values of defined types.

Data Declarations (3)

Values of new types can be used in the same ways as those of built in types. E.g., given:

data Answer = Yes | No | Unknown

we can define:

answers :: [Answer]
answers = [Yes, No, Unknown]

flip :: Answer -> Answer
flip Yes = No
flip No = Yes
flip Unknown = Unknown
Recursive Types (1)

In Haskell, new types can be declared in terms of themselves. That is, types can be **recursive**:

```haskell
data Nat = Zero | Succ Nat
```

`Nat` is a new type with constructors

- `Zero :: Nat`
- `Succ :: Nat -> Nat`

Effectively, we get both a new way form terms and typing rules for these new terms.

Recursion and Recursive Types

Using recursion, it is easy to define functions that convert between values of type `Nat` and `Int`:

```haskell
nat2int :: Nat -> Int
nat2int Zero = 0
nat2int (Succ n) = 1 + nat2int n

int2nat :: Int -> Nat
int2nat 0 = Zero
int2nat n | n >= 1 = Succ (int2nat (n - 1))
```

Recursive Types (2)

A value of type `Nat` is either `Zero`, or of the form `Succ n` where `n :: Nat`. That is, `Nat` contains the following infinite sequence of values:

- `Zero`
- `Succ Zero`
- `Succ (Succ Zero)`

Parameterized Types

Types can also be parameterized on other types:

```haskell
data List a = Nil | Cons a (List a)
```

Resulting constructors:

- `Nil :: List a`
- `Cons :: a -> List a -> List a`

```haskell
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

Resulting constructors:

- `Leaf :: a -> Tree a`
- `Node :: Tree a -> Tree a -> Tree a`