Introduction

Concepts addressed/touched in this unit:

- Curry interpreter PAKCS
- Predefined Types
- Defining Functions
- Defining Types
- Using Lists
- Higher-order Functions
- Conditions and Cases
- Non-determinism
- Variables
- Functional Patterns
- Default Rules
- List comprehensions

Curry interpreter

Functional logic programs are often executed by an interactive *interpreter*. Currently, a popular interpreter for Curry is PAKCS. The trace of an invocation of PAKCS follows:

```
[antoy@localhost INTRO]$ pakcs

     some omitted lines

Type ":h" for help


Prelude>
```

The user can issue *commands*, such as `:quit` to terminate, or *expressions* to evaluate, such as `2+2`.

```
Prelude> 2+2
Result: 4 ? ;
No more solutions.
Prelude> :quit
[antoy@localhost INTRO]$
```

The semicolon after the question mark asks for additional results. Functional computations always have one result only.

Several types and functions are available to the user in the interpreter. See the `Prelude.curry` file in the `lib` directory of the PAKCS distribution.

Predefined Types

Several *types* are predefined or built-in.

`Int`	integers:	`...-2,-1,0,1,2,...`
`Bool`	booleans:	`True,False`
`Char`	characters:	`'h','i','\n'`
(α,β)	tuples:	`(1,True), ('J',"Doe")`
$[\alpha]$	lists:	`[],1:2:[],['a','b','c']`
`String`	strings:	`"hello","word"`
`Success`	Success:	no visible values
`Unit`	Unit:	`()`

`Int` is the builtin integer numeric type in PAKCS, it has unlimited precision.

`String` is a synonym of `[Char]`.

`Success` is now deprecated, it may appear in old code as the type of constrains.

`Unit` is a type whose only value is `()`.

Both α and β stand for an arbitrary type. Tuples and lists are polymorphic.

Several common operations are available on these types. See the `Prelude.curry` file in the `lib` directory.

The programmer can define other types referred to as algebraic.

Defining Functions

The programmer can define *functions* in a file, say *myfile*`.curry`, and load them in the interpreter with the command `:load` *myfile*. E.g.:

```
square i = i * i
average (x,y) = (x+y) `div` 2
swap (x,y) = (y,x)
mylen l = if l==[]
            then 0
            else 1 + mylen (tail l)
nine = 9
```

The application of a function (symbol) to its argument(s) is denoted by *juxtaposition*. E.g., the function `square` takes one argument and returns its square.

No type declaration is necessary for the argument(s).

The argument of the function `average` is a pair.

The function `div`, integer division, is optionally used as an infix operation by enclosing it in backquotes.

The function `swap` is polymorphic, it swaps pairs of any types.

The `if` · `then` · `else` · construct has the usual meaning.

Functions can have zero arguments, see `nine`. They are constants.

Defining Types

User-defined types are introduced by a *data* type declaration.

```
data Color = Red | Green | Blue
data MyBool = MyFalse | MyTrue
data Natural = Zero | Succ Natural
data MyList a = Nil | Cons a (MyList a)
data BinTree a
    = Leaf
    | Branch a (BinTree a) (BinTree a)
```

Functions on user-defined types (and predefined as well) are conveniently coded by *pattern matching.*

```
myNot MyFalse = MyTrue
myNot MyTrue = MyFalse
leq Zero _ = True
leq (Succ _) Zero = False
leq (Succ n) (Succ m) = leq n m
lookup _ Leaf = False
lookup a (Branch b left right) =
   a == b ||
   lookup a if a < b then left else right
```

The underscore symbol "_" denotes an *anonymous* variable. A variable that is not used in the result of a function.

Using Lists

Lists are ubiquitous for programming. Here are some simple examples of notations and techniques.

`[2..]` → `[2,3,4,`$\cdots$ infinite
`[2..5]` → `[2,3,4,5]`
`[2,4..12]` → `[2,4,6,8,10,12]`
`[2,4..]` → `[2,4,6,`$\cdots$ infinite

The following example shows *nested* functions and *lazy evaluation.*

```
fibseq = fsaux 0 1
   where fsaux x y = x : fsaux y (x+y)
fibo n = fibseq !! n
test = take 20 fibseq
```

The function (constant) `fibseq` is the sequence of all Fibonacci numbers. It can't be printed (it is infinite), but it can be used.

It uses a nested function defined within a `where`. A nested function can be referenced only by the immediately nesting function and the functions in its block. The variables of the nesting functions can be referenced by the recursively nested functions.

More on Functions (1)

Functions can take other *functions* as arguments. This is a powerful feature to parameterize a computation with another computation.

```
map _ [] = []
map f (x:xs) = f x : map f xs
foldr _ z [] = z
foldr f z (x:xs) = f x (foldr f z xs)
```

The following equations explain what the above functions compute. In both cases, `f` is the function parameter.

```
map f [x1,x2,...,xn] = [f x1, f x2, ... f xn]
foldr f z [x1,x2,...,xn] = (x1 `f` (x2 `f` ... (xn `f` z)...))
```

A function can be defined by an expression, like the expression `2+3` defines a number. A function defined by an expression, referred to as lambda abstraction, is called *anonymous*. For example:

```
sumList l = foldr (+) 0 l

maxList (x:xs) =
  foldr (\u v -> if u>v then u else v) x xs

atoi str = num
  where
    digits = map (\x -> ord x - ord '0') str
    num = foldl (\x y -> 10*x+y) 0 digits
```


More on Functions (2)

Functions can have optional *conditions:*

```
max x y | x < y = y
        | otherwise = x
lookup _ Leaf = False
lookup a (Branch b left right)
    | a < b = lookup a left
    | a > b = lookup a right
    | otherwise = True
```

Case expressions are allowed in the returned value. This is somewhat equivalent the the earlier example.

```
mylen l =
   case l of
        [] -> 0
        (_:xs) -> 1 + mylen (tail l)
```

Both conditions and cases are tested sequentially, from top to bottom. The arm of the first that succeeds is evaluated. Otherwise the evalution fails.

More on Types

The user can optionally declare the type of a symbol and/or ask the interpreter to show the type of a symbol.

Functions take "one argument only at the time." The type of a function that takes an argument of type α and returns an argument of type β is α -> β.

Referring to previous examples:

```
myNot :: MyBool -> MyBool
leq :: Natural -> Natural -> Bool
lookup :: Int -> BinTree Int -> Bool
```

The operator -> is right-associative which means that function lookup takes an Int and returns a function of type BinTree Int -> Bool. Thus, this function takes a BinTree Int and returns a Bool.

This style of function definition is called *curried*. By contrast:

```
average :: (Int,Int) -> Int
```

i.e., the function average takes a pair of Int and returns an Int.

To get the interpreter tell you the type of a symbol, say s, load the file defining s and input the command :type s.

Non-Determinism (1)

Function (and constants) may return more than one value. E.g., in file sample4.curry:

```
coin = 0
coin = 1
```

The constant coin has two values, 0 and 1. If it appears in a computation, one of its values is choosen (don't count on it as a random bit generator).

```
sample4> coin+6
Result: 6 ? ;
Result: 7 ? ;
No more solutions.
sample4> coin==coin
Result: True ? ;
Result: False ?
sample4>
```

Non-deterministic computations are convenient when the programmer does not have a better algorithm for making a choice, e.g., which move to make in a puzzle.

Or when the programmer does not want to put the effort and time to learn, code, and test an algorithm for making a choice.

It is intended that a non-deterministic choice is later *constrained* to filter out "bad" choices.

Non-Determinism (2)

Compute a *permutation* of a list.
Every permutation should be produced.
Use the testing properties tool to verify the code.

```
import Test.Prop

perm []         = []
perm (xs++x:ys) = x : perm (xs++ys)

sample = [1,2,3,4]

-- the reverse of the input is produced
prop_1 = perm sample ~> reverse sample

-- some random permutation is produced
prop_2 = perm sample ~> [2,4,1,3]

-- the number of perms is the factorial
prop_3 = perm sample # 4*3*2*1
```

To understand the code, draw a diagram of input and output.

Execute with curry check *filename*.

Variables

A variable in the left-hand side of a rule matches a subexpression of an argument.

A variable in the right-hand of a rule is *shared*:

```
coin = 0 ? 1
double x = x + x
t = coin + coin
u = double coin
```

A bound (left-hand side) variable stands for *any* expression. A free variable stands for *some* expression that produces a result.

```
data Color = Red | Green | Blue
f Red = 1
f Green = 2
t = (f x, x) where x free
```

Variables in the left-hand side of a where equation are names of subexpressions.

```
f x = y * y
    where y = x - 1
```

List comprehensions

A special notation is available to transform lists. Here are some examples:

```
triangNumbers =
   [x * (x+1) `div` 2 | x <- [0..]]
primeNumbers =
   [x | x <- [2..], isPrime x]
isPrime x =
   and [x `mod` y /= 0 | y <- [2..x `div` 2]]
lexPairs =
   [(x,y) | x <- [0..3], y <- [x..3]]
```

The part of the form *var* <- *list* is called a *generator*.

A condition following generators is called a *guard* and it acts as a filter.

Multiple generators are allowed and lexcally scoped.

```
qsort :: [Int] -> [Int]
qsort [] = []
qsort (x:l) = qsort [y | y <- l, y<x] ++
              x:qsort [z | z <- l, z>=x]
```


Exercises (optional/proposals)

Exercise I1. Define the *absolute value* function.

Exercise I2. Define a function that counts how many integers in a list are even, and run it on a few test samples. Hint: the *remainder* function is `mod` (similar to `div`).

Exercise I3. Define a function that counts how many integers in a list are positive, and run it on a few test samples. Observe the similarities with the previous function.

Exercise I4. Define a function, say `gencount`, that counts how many elements in a list have a certain property. Redefine the previous functions using `gencount`.

Exercise I5. Define a type to represent a tree. A tree has a root and zero or more children which are trees. Most often, the root has a decoration.

Exercise I6. Build a *trie* from a list of words and use it to find whether a word is in a dictionary. Use the previous exercise.