Part I: Functional Programming

- A pure functional program consists of data, functions, and an expression which describes a result.
- Missing: variables, assignment, side-effects.
- A processor of a functional program is essentially a calculator.

Example: (transcript of a session with funny, the Funnel interpreter)

```
/home/odersky/tmp> funny
> def gcd (a, b) = if (b == 0) a else gcd (b, a % b)
'def gcd'
> gcd (8, 10)
2
> val x = gcd (15, 70)
'val x = 5'
> val y = gcd (x, x)
'val y = 5'
```


Why Study Functional Programming?

- FP is programming in its simplest form ⇒ easier to understand thoroughly than more complex variants.
- FP has powerful composition constructs.
- In FP, one the *value* of an expression matters since side effects are impossible. (this property is called *referential transparency*).
- Referential transparency gives a rich set of laws to transform programs.
- FP has a well-established theoretical basis in Lambda Calculus and Denotational Semantics.

A Second Example: Square Roots by Newton's Method

Compute the square root of a given number x as a limit of the sequence y_i given by:

$$
\begin{aligned}
y_0 &= 1 \\
y_{i+1} &= (y_i + x/y_i)/2
\end{aligned}
$$

The $i \rightarrow i + 1$ step is encoded in the function improve:

```
> def improve (guess, x) = (guess + x / guess) / 2
> val y0 = 1.0
> val y1 = improve (y0, 2.0)
'val y1 = (1.5)'
> val y2 = improve (y1, 2.0)
'val y2 = (1.4166667)'
> val y3 = improve (y2, 2.0)
'val y3 = (1.4142157)'
```


We have to stop the iteration when the result is good enough:

```
> def goodEnough (guess, x) = abs ((guess * guess) – x) < 0.001f
'def goodEnough'
> def sqrtIter (guess, x) = {
|    if (goodEnough (guess, x)) guess
|    else sqrtIter (improve (guess, x), x)
| }
'def sqrtIter'
> def sqrt (x) = sqrtIter (1.0, x)
'def sqrt'
> sqrt (2.0)
1.4142157
```

Language Elements Seen So Far

- Function Definitions:

  def <ident> parameters "=" expression
- Value definitions:

  val <ident> "=" expression
- Function application: ident expression
- Tuples: (expr$_1$, ..., expr$_2$)
- Numbers, operators: as in Java
- If-then-else: as in Java.
- Grouping by parentheses "(" ")" or braces "{" "}".
- Indentation is significant at top level and between braces "{" "}".
- No static type information; types are maintained at run time.

Nested Functions

In the previous example we defined functions improve, goodEnough, sqrtIter and sqrt.

If functions are used only internally by some other function we can avoid "name-space pollution" by nesting. E.g:

```
def sqrt (x) = {
   def improve (guess, x) = (guess + x / guess) / 2
   def goodEnough (guess, x) = abs ((guess * guess) - x) < 0.001f
   def sqrtIter (guess, x) =
      if (goodEnough (guess, x)) guess
      else sqrtIter (improve (guess, x), x)

   sqrtIter (1.0, x)
}
```

The visibility of an identifier extends from its own definition to the end of the enclosing block, including any nested definitions.

A Note on Indentation:

- Indentation is significant inside "{" and "}".
- We can change "{" and "}" to "(" and ")", but then we have to write semicolons between definitions and statements:

```
def sqrt (x) = (
   def improve (guess, x) = (guess + x / guess) / 2 ;
   def goodEnough (guess, x) = abs ((guess * guess) - x) < 0.001f ;
   def sqrtIter (guess, x) =
      if (goodEnough (guess, x)) guess
      else sqrtIter (improve (guess, x), x) ;

   sqrtIter (1.0, x)
)
```


Exercise:

The goodEnough function tests the absolute difference between the input parameter and the square of the guess.

This is not very accurate for square roots of very small numbers and might lead to divergence for very large numbers (why?).

Design a different sqrtIter function which stops if the *change* from one iteration to the next is a small fraction of the guess. E.g.

$$abs((x_{i+1} - x_i)/x_i) < 0.001$$

Complete:

```
def sqrtIter (guess, x) = ?
```

Semantics of Function Application

One simple rule: A function application f (A) is evaluated by

- replacing the application with the function's body where,
- actual parameters A replace formal parameters of f.

This can be formalised as a *rewriting of the program itself*:

def f (x) = B ; ... f (A) $\rightarrow$ **def** f (x) = B ; ... [A/x] B

Here, [A/x] B stands for B with all occurrences of x replaced by A.

[A/x] B is called a *substitution*.

Rewriting Example:

Consider gcd:

def gcd (a, b) = **if** (b == 0) a **else** gcd (b, a % b)

Then gcd (14, 21) evaluates as follows:

	gcd (14, 21)
$\rightarrow$	**if** (21 == 0) 14 **else** gcd (21, 14 % 21)
$\rightarrow$	gcd (21, 14)
$\rightarrow$	**if** (14 == 0) 21 **else** gcd (14, 21 % 14)
$\rightarrow$	gcd (14, 21 % 14)
$\rightarrow$	gcd (14, 7)
$\rightarrow$	**if** (7 == 0) 14 **else** gcd (7, 14 % 7)
$\rightarrow$	gcd (7, 14 % 7)
$\rightarrow$	gcd (7, 0)
$\rightarrow$	**if** (0 == 0) 7 **else** gcd (0, 7 % 0)
$\rightarrow$	7

Another rewriting example:

Consider factorial:

def factorial (n) = **if** (n == 0) 1 **else** n * factorial (n – 1)

Then factorial(5) rewrites as follows:

	factorial (5)
$\rightarrow$	**if** (5 == 0) 1 **else** 5 * factorial (5 – 1)
$\rightarrow$	5 * factorial (5 – 1)
$\rightarrow$	5 * factorial (4)
$\rightarrow$ ... $\rightarrow$	5 * (4 * factorial (3))
$\rightarrow$ ... $\rightarrow$	5 * (4 * (3 * factorial (2)))
$\rightarrow$ ... $\rightarrow$	5 * (4 * (3 * (2 * factorial (1))))
$\rightarrow$ ... $\rightarrow$	5 * (4 * (3 * (2 * (1 * factorial (0))))
$\rightarrow$ ... $\rightarrow$	5 * (4 * (3 * (2 * (1 * 1))))
$\rightarrow$ ... $\rightarrow$	120

What differences are there between the two rewrite sequences?

Tail Recursion

Implementation note: If a function calls itself as its last action, the function's stack frame can be re-used. This is called "tail recursion".

$\Rightarrow$ Tail-recursive functions are iterative processes.

More generally, if the last action of a function is a call to another (possible the same) function, only a single stack frame is needed for both functions. Such calls are called "tail calls".

Exercise: Design a tail-recursive version of factorial.

First-Class Functions

Most functional languages treat functions as “first-class values”.

That is, like any other value, a function may be passed as a parameter or returned as a result.

This provides a flexible mechanism for program composition.

Functions which take other functions as parameters or return them as results are called “higher-order” functions..

Example:

Sum integers between a and b:

```
def sumInts (a, b) =
   if (a > b) 0 else a + sumInts (a + 1, b)
```

Sum cubes of all integers between a and b:

```
def cube (a) = a * a * a
def sumCubes (a, b) =
   if (a > b) 0 else cube (a) + sumCubes (a + 1, b)
```

Sum reciprocals between a and b

```
def sumReciprocals (a, b) =
   if (a > b) 0 else 1.0 / a + sumReciprocals (a + 1, b)
```

These are all special cases of $\sum_a^b f(n)$ for different values of f.

Can we factor out the common pattern?

Summation with a higher-order function

Define:

```
def sum (f, a, b) = {
   if (a > b) 0
   else f (a) + sum (f, a + 1, b)
}
```

Then we can write:

```
def sumInts (a, b) = sum (id, a, b)
def sumCubes (a, b) = sum (cube, a, b)
def sumReciprocals (a, b) = sum (reciprocal, a, b)
```

where

```
def id (x) = x
def cube (x) = x * x * x
def reciprocal (x) = 1.0 / x
```


Anonymous functions

- Parameterisation by functions tends to create many small functions.
- Sometimes it is cumbersome to have to define the functions using **def**.
- A shorter notation makes use of *anonymous functions*, defined as follows:

  (x_1, ..., x_n | E) defines a function which maps its parameters x_1, ..., x_n to the result of the expression E (where E may refer to x_1, ..., x_n).
- Anonymous functions are not essential in Funnel; an anonymous function (x_1, ..., x_n | E) can always be expressed using a **def** as follows:

  (**def** f (x_1, ..., x_n) = E ; f)

  where f is fresh name which is used nowhere else in the program.
- We also say, anonymous functions are “syntactic sugar”.

Summation with Anonymous Functions

Now we can write shorter:

```
def sumInts (a, b) = sum ((x | x), a, b)
def sumCubes (a, b) = sum ((x | x * x * x), a, b)
def sumReciprocals (a, b) = sum ((x | 1.0 / x), a, b)
```

Can we do even better?

Hint: a, b appears everywhere and does not seem to take part in interesting combinations. Can we get rid of it?

Currying

Let's rewrite sum as follows.

```
def sum (f) = {
  def sumFun (a, b) =
    if (a > b) 0
    else f (a) + sumFun (a + 1, b)
  sumFun
}
```

- sum is now a function which returns another function, namely the specialized summing function which applies the f function and sums up the results. Then we can define:

```
val sumInts = sum (x | x)
val sumCubes = sum (x | x * x * x)
val sumReciprocals = sum (x | 1.0 / x)
```

- Function values can be applied like other functions:

```
sumReciprocals (1, 1000)
```


Curried Application

How are function-returning functions applied?

Example:

```
> sum (cube) (1, 10)
3025
```

- sum (cube) applies sum to cube and returns the "cube-summing function" (Hence, sum (cube) is equiavlent to sumCubes).
- This function is then applied to the pair (1, 10).
- Hence, function application associates to the left:

```
sum (cube) (1, 10)  ==  (sum (cube)) (1, 10)
                    ==  val sc = sum (cube) ; sc (1, 10)
```

- As a convenience, we can also drop the parentheses around one-symbol arguments, e.g. sum cube (1, 10)

Curried Definition

The style of function-returning functions is so useful in FP, that we have special syntax for it.

For instance, the next definition of sum is equivalent to the previous one, but shorter:

```
def sum f (a, b) = {
  if (a > b) 0
  else f a + sum f (a + 1, b)
}
```

Generally, a curried function definition

def f (args$_1$) ... (args$_n$) = E

where $n > 1$ expands to

def f (args$_1$) ... (args$_{n-1}$) = (**def** g (args$_n$) = E ; g)

where g is a fresh identifier. Or, shorter:

def f $(\text{args}_1) \ldots (\text{args}_{n-1}) = (\text{ args}_n \mid E\ )$

Performing this step n times yields that

def f $(\text{args}_1) \ldots (\text{args}_{n-1})\ (\text{args}_n) = E$

is equivalent to

def f $= (\text{args}_1 \mid (\text{ args}_2 \mid \ldots (\text{ args}_n \mid E\ ) \ldots))$

- Again, parentheses around single-name formal parameters may be dropped.
- This style of function definition and application is called *currying* after its promoter, Haskell B. Curry.
- Actually, the idea goes back further to Frege and Schönfinkel, but the name “curried” caught on (maybe because “schönfinkeled” does not sound so well.)

Exercises:

1. The sum function uses a linear recursion. Can you write a tail-recursive one by filling in the ??'s?

```
def sum f (a, b) = {
    def iter (a, result) = {
        if (??) ??
        else iter (??, ??)
    }
    iter (??, ??)
}
```

2. Write a function product that computes the product of the values of functions at points over a given range.

3. Write factorial in terms of product.

4. Can you write an even more general function which generalizes both sum and product?

Functions and Data

In this sections we will learn how functions create and encapsulate data structures.

Example: Rational Numbers

We want to design a package to do rational arithmetic. Let us postulate three base functions:

- makeRat (n, d) returns the rational n/d with numerator n and denominator d.
- numer x returns the numerator of the rational number x.
- denom x returns the denominator of x.

Then we can define arithmetic functions on rationals which implement the standard rules:

$$\begin{aligned}
\frac{n_1}{d_1} + \frac{n_2}{d_2} &= \frac{n_1 d_2 + n_2 d_1}{d_1 d_2} \\
\frac{n_1}{d_1} - \frac{n_2}{d_2} &= \frac{n_1 d_2 - n_2 d_1}{d_1 d_2} \\
\frac{n_1}{d_1} \cdot \frac{n_2}{d_2} &= \frac{n_1 n_2}{d_1 d_2} \\
\frac{n_1}{d_1} / \frac{n_2}{d_2} &= \frac{n_1 d_2}{d_1 n_2} \\
\frac{n_1}{d_1} = \frac{n_2}{d_2} &\quad \text{iff} \quad n_1 d_2 = d_1 n_2
\end{aligned}$$

Example:

```
def addRat (x, y) = makeRat (
    numer (x) * denom (y) + numer (y) * denom (x),
    denom (x) * denom (y))
```

etc.

Here's a definition of the base functions for rationals:

```
def makeRat (x, y) = (x, y)
def numer (x, y) = x
def denom (x, y) = y
```

Let's test this, by rewriting denom (makeRat (1, 2)):

```
denom (makeRat (1, 2)) = denom (1, 2) = 2
```

Note that tuples can be not only arguments but also results of functions.

Exercise

The rational functions do not reduce rationals to lowest terms. Example:

```
> val x = (makeRat (2, 3)
'val x = ((2, 3))'
> val y = makeRat (1, 3)
'val y = ((1, 3))'
> val z = addRat (x, y)
'val z = ((9, 9))'
```

Rewrite function makeRat so that it reduces rationals to lowest terms.

Records

- The rational number abstraction is not very tight, since clients can still create and access the pairs representing rational numbers directly, without going through makeRat, numer and denom.
- We can provide better documentation and robustness by having makeRat return a *record* rather than a tuple. Example:

```
def makeRat (x, y) = {
   def numer () = x
   def denom () = y
}
```

- This returns a record with two named functions, numer and denom.
- Generally, if a (...) or {...} block ends in a definition, this block is assumed to be a record which has as fields all names defined between the parentheses or braces.

  ⇒ Definitions are now also first class values.

Record Access

Methods in a record are accessed with the usual dot notation. Example:

```
def addRat (x, y) = makeRat (
   x.numer () * y.denom () + y.numer () * x.denom (),
   x.denom () * y.denom ())

def dispRat (x) = x.numer () + "/" + x.denom ()
```

Functions without parameters

- Records with simple accessors create a lot of functions with () parameters. Applications of such functions are often tedious to write.
- As a convenience, Funnel also lets you write functions without any parameters.
- These functions will be evaluated every time they are used.
- Examples:

```
def makeRat (x, y) = {
  def numer = x
  def denom = y
}
def addRat (x, y) = makeRat (
  x.numer * y.denom + y.numer * x.denom,
  x.denom * y.denom )
def dispRat (x) = x.numer + "/" + x.denom
```


- As a consequence, parameterless functions cannot be used themselves as function parameters. Given

```
def f () = E
def g = E
```

The expression h (f) would pass the *function* f, whereas h (g) would pass the *result* of evaluating g.

If we wanted to pass g as a function, we'd have to use the anonymous function (|g).

- A useful law: For any expression E:

```
E = (|E) ()
```

Question: What is the difference between a parameterless function

```
def a = gcd (x, y)
```

and the following value definition?

```
val a = gcd (x, y)
```


Local Values

If we implement the "reduce to lowest terms" version makeRat with records, we might get:

```
def makeRat (x, y) = {
  val g = gcd (x, y)
  def numer = x / g
  def denom = y / g
}
```

This is less efficient than one would like. Why?

Better efficiency is attained by using **val** definitions to evaluate the intermediate expressions:

```
def makeRat (x, y) = {
  val g = gcd (x, y) ; val n = x / g, d = y / g
  def numer = n
  def denom = d
}
```


Objects

We can go further and also package functions operating on a data abstraction with the data abstraction itself.

Example: Rational numbers would now in addition to functions numer and denom have functions add, sub, mul, **div**, equal, toString.

An implementation could be as follows:

```
def makeRat (x, y) = {
  def numer = x
  def denom = y
  def add (y) = makeRat (
    numer * y.denom + y.numer * denom,
    denom * y.denom)
  def sub (y) = ...
  ...
  def toString = numer + "/" + denom
}
```

Here is a client of the new rational abstraction:

```
val x = makeRat (1, 3)
val y = makeRat (5, 7)
val z = makeRat (3, 2)
x.add (y).mul (z).toString
```

This evaluates $(1/3 + 5/7) * 3/2$ and converts to string.

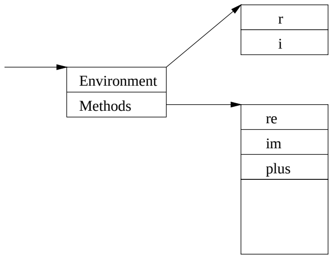
Implementation Considerations

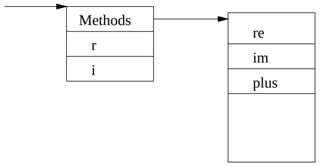
- When returning a function, or a record of functions, the returned functions can access parameters or local variables of their enclosing function.

  ⇒ These parameters and local variables cannot be stored on the stack; they must be stored in the heap.
- We usually group them together in an *environment*.
- A record then consists of an environment and a method tuple.
- The environment can be different for every record we create.
- The method tuple, on the other hand, does not change from one creation to the next.

  ⇒ Can share a single tuple among all records created by a function.

Object Representations





Environment	~	Instance Variables
Fields	~	Method Table
Record Expression	~	Class
Record Value	~	Object

Summary

- We have seen an introduction to Funnel.
- Main syntactic elements:
  - Function definitions with **def**.
  - Value definitions with **val**.
  - Definitions can be nested.
  - Functions are first class ⇒ higher-order functions.
  - Definitions are first class ⇒ records.
  - Analogy records – objects,
  - Analogy record expressions – classes.
- This will be deepened and formalized in the following lectures.