Programmation IV

.. is concerned with:

advanced (and not-yet completely standard) programming paradigms.

In particular:

- Functional Programming
- Logic Programming

Programming Paradigms

Paradigm: "An example that serves as pattern or model".

In this course: A pattern that serves as a *school of thoughts* for programming of computers.

Main paradigms:

- Imperative Programming
- Functional Programming
- Logic Programming
- Orthogonal to above: Object-Orientation.

Imperative Programming

Programming with:

- (Mutable) variables,
- Assignments
- Control-structures such as:

  If-then-else, loops, break, contine, return, goto.

Two possible ways of conceptualizing of a program:

- As an instruction sequence for a Van Neumann Machine
- As a relation between predicates.

Programs as Instruction Sequences for Van Neumann Machines

Variable	=	memory cell
Variable reference	=	load
Variable assignment	=	store
Control structures	=	jumps

Problem: Scalability – need to describe operations one word at a time.

Example: Linear Search

Define an invariant:

$$I \equiv i \leq a.length \wedge \forall j.0 \leq j < i.a(j) \neq x$$

Then:

$\{\ true\ \}$
var $i := 0$
$\{\ I\ \}$
$while\ (i < a.length\ \&\&\ a(i)\ != x)$
$\{\ i < a.length \wedge a(i) \neq x \wedge \forall j.0 \leq j < i.a(j) \neq x\ \}$
$i = i + 1$
$\{\ I\ \}$
$\{\ I \wedge \neg(i < a.length \wedge a(i) \neq x)\ \}$

The last predicate simplifies to:

$$I \wedge i = a.length \vee a(i) = x.$$

Example

Swapping two variables:

$$\{y = Y, x = X\}$$
$$t := x$$
$$\{y = Y, t = X\}$$
$$x := y$$
$$\{x = Y, t = X\}$$
$$y := t$$
$$\{x = Y, y = X\}$$

Laws of Imperative Programming

$$\{P\}\ skip\ \{P\}$$

$$\{[e/x]Q\}\ x := e\ \{Q\}$$

$$\frac{\{P \wedge c\}\ s_1 \{Q\} \qquad \{P \wedge \neg c\} s_2\ \{Q\}}{\{P\}\ \textbf{\textit{if}}\,(c)\ s_1\ \textbf{\textit{else}}\ s_2\ \{Q\}}$$

$$\frac{\{P \wedge c\ \} s\ \{P\}}{\{P\}\ \textbf{while}\ (c)\ s\ \{P \wedge \neg c\}}$$

$$\frac{P' \Rightarrow P \qquad \{P\}\ s\ \{Q\} \qquad Q \Rightarrow Q'}{\{P'\}\ s\ \{Q'\}}$$

(P is loop invariant in rule for *while*).

Programs as Predicate Transformers

- A program relates a precondition with a postcondition.
- Conditions are predicates, i.e. boolean formulas over variables.
- Two versions:
  - Program establishes postcondition after run, provided precondition is met before run (total correctness).
  - Program establishes postcondition after run, provided precondition is met before run and program terminates (partial correctness; termination has to be proved separately).

E.g.: The theory of polynomials defines the sum of two polynomials, with laws such as:

$$(a*x + b) \ + \ (c*x + d) \ = \ (a+c) * x + (b+d)$$

It does not define an operator to change a coeficient in a polynomial while keeping the polynomial the same!

E.g.: The theory of strings defines a concatenation operation *concat* on strings, with laws such as:

$$\text{"abc"}\ concat\ \text{"xyz"} \ = \ \text{"abcxyz"}$$

It does not define an operator to change a letter in a string while keeping the identity of the string the same!

Consequence for scalable software:

- Concentrate on defining theories of new operators.
- Avoid (or: defer) state changes.
- Operators are functions, which are often composed from simpler functions.

Consequences

Imperative programming is constrained by the “Van Neumann” principle.

Need other techniques to define higher-level data such as polynomials, collections, geometric shapes, strings, texts.

Need to define theories of polynomials, collections, texts, etc.

A theory consists of one or more data types with operations which are related by laws.

The theory usually does not describe mutation.

Does object-oriented programming help?

In fact, the opposite is true.

OOP invalidates the imperative laws of programming due to aliasing.

Example:

$$\{\ q.x = 1\ \}\ p.x := 2\ \{\ q.x = 1\ \}$$

This seems to hold as a consequence of the law for assignment.

But this argument is invalid if p and q are aliases of each other!

Hence, the law of assignment is no longer valid if aliases are permitted.

Therefore, programs that mix state-changes with aliasing are very hard to understand and verify.

Assessment of Predicate Transformers

Advantage: Mathematical Precision; can be mechanized.

Problem: Scalability

In the canoncial version of predicate tranformers,

- programs consist of simple variables and arrays,
- no user-defined data staructures,
- no encapsulation.

History of FPL's

1960	Lisp
1975–77	ML, FP, Scheme,
1978	Smalltalk
1985	Miranda
1986	Standard ML
1990	Haskell
1990	Erlang
1996	Caml
1996	Pizza
1997	Java 1.1
2000	OCaml

Some Functional Programming Languages

In the restricted sense:

- Pure Lisp,
- Haskell (without the I/O monad)

In the general sense:

- Lisp, Scheme,
- ML, Haskell,
- Pizza, Scala,
- Smalltalk(!),
- Java(?)

Functional Programming Languages

In the restricted sense, a functional programming language (FPL) is a programming language without mutable variables, assignments, or imperative control structures.

In the general sense, a functional programming language is a language which makes it convenient and elegant to program with functions.

In particular, functions in a FPL are *first-class citizens*. This means:

- Like other data, they can be defined inside other functions.
- Like other values, they can be passed as parameters and returned as results.
- As for other values, there is a set of basic operations that compose functions.

Functional Programming

In the *restricted sense*, functional programming (FP) is programming without variables, assignments, or imperative constrol structures.

In the *general sense*, functional programming is programming with an emphasis on functions.

In particular, functions become values which are produced, consumed, and composed by programs.

This is done easiest in a functional programming language.

Recommended Books

- Structure and Interpretation of Computer Programs. Harold Abelson and Gerald J. Sussman. 2nd edition. MIT Press 1996.
- Approche Fonctionnelle de la Programmation. Guy Cousineau and Michel Mauny. Ediscience International. 1998.
- Introdcution to Functional Programming using Haskell. Richard Bird. Prentice Hall 1998.
- Foundations of Logic Programming. J.W. Lloyd. Springer Verlag. 1984.
- Effective Java. Joshua Bloch. Addison Wesley, 2001.

Many examples and exercises in the course are taken from Abelson and Sussman.

But instead of Scheme we use...

Scala

In this course we teach FP with a new language: Scala.

Scala has all the constructs of a typical modern FPL, such as:

- first-class functions,
- pattern matching,
- flexible static safe typing.

Scala is also a (pure) object-oriented language.

Scala interoperates seamlessly with Java.

Elements of Programming

Every non-trivial programming language has:

- Primitive expressions, which represent the simplest entities the language is concerned with.
- Means of combination, by which elements are built from simpler ones.
- Means of abstraction, by which elements are named and manipulated as units.

Expressions

Functional programming is a bit like using a calculator.

The user types expressions and the system responds by computing their value.

Example:

```
? 87 + 145
232

? 1000 - 333
667

? 5 + 2 * 3
11
```

Example

Here is an evaluation of an arithmetic expression.

$$
\begin{array}{ll}
 & (2 * \mathit{pi}) * \mathit{radius} \\
\rightarrow & (2 * 3.14159) * \mathit{radius} \\
\rightarrow & 6.28318 * \mathit{radius} \\
\rightarrow & 6.28318 * 10 \\
\rightarrow & 62.8318
\end{array}
$$

The process of stepwise simplification of expressions to values is called reduction.

Evaluation

A compound expression is evaluated by repeatedly applying the following simplification steps.

- pick the left-most operation
- evaluate its operands
- apply the operator to the operand values.

A defined name is evaluated by replacing the name by th defintion's right hand side.

The evaluation process stops once we have reached a value.

A value is a number (for now).

Later on, we will get to know more complicated kinds of values.


```
? def circumference = 2 * pi * radius
def circumference: Double

? circumference
62.8318
```

The Scala system keeps track of definitions in an environment, which is a table of name-value pairs.

The Environment

Functional programming languages are richer than calculators, because they also allow to define new quantities and combinators.

Example:

```
? def size = 2
def size: Int

? 5 * size
10

? def pi = 3.14159
def pi: Double

? def radius = 10
def radius: Int

? 2 * pi * radius
62.8318
```

Parameters

Definitions can define functions with parameters.

Example:

? **def** $square(x\colon Double) = x * x$
def $square(x\colon Double)\colon Double$

? $square(2)$
4.0

? $square(5 + 4)$
81.0

? $square(square(4))$
256.0

? **def** $sumOfSquares(x\colon Double,\ y\colon Double) = square(x) + square(y)$
def $sumOfSquares(x\colon Double,\ y\colon Double)\colon Double$

Evaluation of Functions

Applications of functions with parameters are evaluated in the same way as operations are:

- evaluate all arguments of the function (in left-to-right order).
- replace the function application by the function's right hand side, and at the same time
- replace all formal parameters of the function by actual arguments.

Example:

$$
\begin{array}{ll}
 & sumOfSquares(3,\ 2+2) \\
\to & \\
 & sumOfSquares(3,\ 4) \\
\to & \\
 & square(3) + square(4) \\
\to & \\
 & 3 * 3 + square(4) \\
\to & \\
 & 9 + square(4) \\
\to & \\
 & 9 + 4 * 4 \\
\to & \\
 & 9 + 16 \\
\to & \\
 & 25
\end{array}
$$

This is called the substitution model of computation.

Call-by-Value vs Call-By-Name

The interpreter reduces function arguments to values before rewriting the function application.

One can instead also apply the function to unreduced arguments.

Example:

```
? def constOne(x: Int, def y: Int) = 1
constOne(x: Int, def y: Int): Int

? constOne(1, loop)
1

? constOne(loop, 2)                    (infinite loop)
^C
```


Consider:

```
? def loop: Int = loop
def loop: Int

? def first(x: Int, y: Int) = x
def first(x: Int, y: Int): Int
```

Then with

call-by-name	call-by-value
first(1, loop)	first(1, loop)
→	→
1	first(1, loop)
	→
	first(1, loop)
	→
	...

Scala uses call-by-value except if the parameter is preceded by ***def***.

The second evaluation order is known as call-by-name, whereas the first one is known as call-by-value.

For expressions that uses only pure functions and that therefore can be reduced with the substitution model, both schemes yield the same final values.

Call-by-value has the advantage that it avoids repeated evaluation of arguments.

Call-by-name has the advantage that it avoids evaluation of arguments when the parameter is not used at all by the function.

Call-by-name evaluation always terminates in a value if call-by-value does.

The reverse is not true!

Example:

```
  sumOfSquares(3, 2+2)
→
  square(3) + square(2+2)
→
  3 * 3 + square(2+2)
→
  9 + square(2+2)
→
  9 + (2+2) * (2+2)
→
  9 + 4 * (2+2)
→
  9 + 4 * 4
→
  9 + 16
→
  25
```

Conditional Expressions and Predicates

Scala offers an ***if–else*** expression to let one choose between two alternatives.

Its syntax is like Java's ***if–else***, but its usage is like Java's conditional expression *... ? ... : ...* .

Example:

```
? def abs(x: Double) = if (x ≥ 0) x else −x
abs(x: Double): Double
```

$x \geq y$ is a predicate of type *Boolean*.

Boolean expressions are formed from the constants *True* and *False*, comparison operators, boolean negation *!* and the boolean operators *&&* and *||*.

Exercise: Define *&&* and *||* in terms of ***if***.

Example: Square Roots by Newton's Method

Task: Write a function

```
def sqrt(x: Double): Double = ... // compute square root of 'x'.
```

The most common way to compute square roots is by Newton's method of successive approximations:

- Start with an initial guess y (say: $y = 1$).
- Improve the guess by taking the average of y and x/y.

Example: *sqrt(2)*

Guess	Quotient	Average
1	2/1 = 2	1.5
1.5	2/1.5 = 1.3333	1.4167
1.4167	2/1.4167 = 1.4118	1.4142
1.4142	...	...

Implementation in Scala

1. Define a function for iterating from a guess to the the result:

```
def sqrtIter(guess: Double, x: Double): Double =
  if (isGoodEnough(guess, x)) guess
  else sqrtIter(improve(guess, x), x)
```

Note that *sqrtIter* is recursive.

Recursive functions need to be provided with an explicit return type in Scala.

For non-recursive functions the return type is optional; if it is missing it is computed from the function's right-hand-side.

2. Define a function to *improve* the guess and a termination test *isGoodEnough*.


```
def improve(guess: Double, x: Double) =
  (guess + x / guess) / 2

def isGoodEnough(guess: Double, x: Double) =
  abs(square(guess) − x) < 0.001
```

3. Define the *sqrt* function.

```
def sqrt(x: Double) = sqrtIter(1.0, x)
```

Exercise: The *isGoodEnough* test is not very precise for small numbers and might lead to non-termination for very large ones (why?).

Design a different version *isGoodEnough* which does not have these problems.

Exercise: Trace the execution of the *sqrt(4)* expression.

Nested Functions

Functional programming encourages the construction of many small helper functions.

The names of functions such as *sqrtIter*, *improve* and *isGodEnough* are relevant only for the implementation of *sqrt*.

We normally do not want users of *sqrt* to acess these functions directly.

We can enforce this (and avoid name-space pollution) by including helper function within the calling function itself:


```
def sqrt(x: Double) = {
  def sqrtIter(guess: Double, x: Double): Double =
    if (isGoodEnough(guess, x)) guess
    else sqrtIter(improve(guess, x), x)

  def improve(guess: Double, x: Double) =
    (guess + x / guess) / 2

  def isGoodEnough(guess: Double, x: Double) =
    abs(square(guess) - x) < 0.001

  sqrtIter(1.0, x)
}
```

Notes:

- { ... } encloses a block.
- The block ends in an expression which defines its value.
- This expression may be preceded by auxiliary definitions.
- Definitions within a block are visible only in the block itself.
- Definitions within a block shadow definitions in outer blocks.

Indentation

If the opening brace of a block is followed by an end-of-line, indentation in the block is significant.

The final expression of the block must then start at the same column as previous definitions. If it starts further right, it is taken as continuation of the previous function definition.

Example:

```
def foo = {                                  def foo = {
  def bar =                                    def bar =
    x            is interpreted as               x(x)
    (x)                                        }
}
```

In place of braces { ... } one can also use parentheses (...) to delimit a block. In that case, indentation is not significant, and the final expression must be preceded by a semicolon.

Example:

```
def foo = (
  def bar =
    x
  ; (x)
)
```

Semicolons may also be used within brace-delimited blocks to make separation into statements clear.

Lexical Scoping

Names defined in outer blocks are visible also in inner blocks, provided they are not shadowed there.

Therefore, we can simplify our *sqrt* example by omitting passing *x* to inner functions.

```
def sqrt(x: Double) = {
  def sqrtIter(guess: Double): Double =
    if (isGoodEnough(guess)) guess
    else sqrtIter(improve(guess))

  def improve(guess: Double) =
    (guess + x / guess) / 2

  def isGoodEnough(guess: Double) =
    abs(square(guess) - x) < 0.001

  sqrtIter(1.0)
}
```