Programming paradigms

Definition

The style/approach used for programming with a PL

Most times a programming paradigm is based on an emerging computational model

Main examples of paradigms

- *imperative* (closer to the underlying hardware model)
  based on the notions of *instruction* and *state*
  - *procedural* (example: C)
  - *object-oriented* (example: Java)
- *declarative* (based on a more abstract model)
  - *functional* (example: ML)
    based on the notions of *function definition* and *function application*
  - *logic* (example: Prolog)
    based on the notions of *logic rule* and *query*

Programming paradigms

Remark

A modern PL embraces several paradigms to favor flexibility

Examples

Java, C#, JavaScript, Python and others support both the imperative (mainly object-oriented, but also procedural) and the declarative paradigm (mainly functional)

Purely functional paradigm

In a nutshell

- program=definitions of mathematical functions + a main expression
- computation=function application (=function call)
- no notion of state: no variable assignment, more in general, no statements, just expressions
- variables=function parameters or local "variables" storing constant values

Functions are *first class values*

Functions can be the result of the evaluation of expressions

Terminology

- *higher order functions*: functions that can accept functions as arguments or/and can return functions
- *lambda expressions/functions or anonymous functions*: functions obtained by the evaluation of an expression

PL and functional programming (FP)

Examples of languages considered primarily functional

- LISP (first functional languages, late 50s)
- ML (early 70s) and its family (OCaml, F#)
- Scheme (mid 70s, derived from LISP)
- Haskell (early 90s, purely functional)
- Clojure (2007, derived from LISP)

Examples of languages supporting FP

- C++
- C#
- Java
- JavaScript
- Kotlin
- Scala
- Python

FP for beginners

What are the PL suggested for FP beginners?

- Hard to tell ...
- Most modern mainstream PL are multi-paradigm and support FP. However
  - not all typical FP features are supported (example: pattern matching)
  - FP cannot be easily isolated, additional features which are not peculiar of FP have to be learned
- there exist easier languages, although they are not mainstream

Why learning FP

- All mainstream languages strongly support FP
- FP is well-suited for several programming styles
  - generic programming, to support code reuse and maintenance
  - event based programming (example: JavaScript/Node.js)
  - concurrent programming (example: Erlang)

OCaml

What is OCaml?

- French dialect of ML (1996)
- Multi-paradigm language with a purely functional core
- Statically typed with type inference
  - type errors detected statically
  - types can be omitted in programs

OCaml core

Syntax

EBNF grammar:

```
Exp ::= ID | NUM | Exp Exp | 'fun' Pat+ '->' Exp | UOP Exp |
    Exp BOP Exp | '(' Exp ')'
Pat ::= ID // simplified pattern
```

Quick comments

- `ID` variable identifiers `[a-zA-Z_][\w']*`
- `NUM` natural numbers

  `0[bB][01][01_]*|0[oO][0-7][0-7_]*|0[xX][0-9a-fA-F][0-9a-fA-F_]*|\d[\d_]*`
- `UOP` unary arithmetic operators `[+-]`
- `BOP` binary arithmetic operators `[+-*/]|mod`
- `Pat` patterns; for simplicity just identifiers

OCaml core

Syntax

EBNF grammar:

```
Exp ::= ID | NUM | Exp Exp | 'fun' Pat+ '->' Exp | UOP Exp |
    Exp BOP Exp | '(' Exp ')'
Pat ::= ID // simplified pattern
```

Functions and application

- examples of anonymous functions

```
fun x -> x+1 (* the increment function *)
fun x y -> x+y (* the addition function *)
```

- function application

```
(fun x -> x+1) 3 (* evaluation returns 4 *)
```

OCaml core

Syntax

EBNF grammar:

```
Exp ::= ID | NUM | Exp Exp | 'fun' Pat+ '->' Exp | UOP Exp |
    Exp BOP Exp | '(' Exp ')'
Pat ::= ID // simplified pattern
```

More on application

`exp1 exp2`

- evaluation of `exp1` is expected to return a function *f*
- evaluation of `exp2` is expected to return a valid argument *a*
- evaluation of `exp1 exp2` returns $f(a)$ (*f* applied to *a*)

OCaml core

Precedence and associativity rules

- standard rules for arithmetic expressions
- application is left-associative

```
(fun x y -> x+y) 3 4   (* is ((fun x y -> x+y) 3) 4 *)
```

- application has higher precedence then binary operators

```
(fun x->x*2) 1+2 (* is ((fun x->x*2) 1)+2 *)
1+(fun x->x*2) 2 (* is 1+((fun x->x*2) 2) *)
```

- anonymous functions have lower precedence then application and binary operators

```
fun x->x*2    (* is fun x->(x*2) *)
fun f a->f a  (* is fun f a->(f a) *)
```

- more critical cases: application and unary operators

```
f + 3 (* addition *)        f (+3) (* application *)
f - 3 (* subtraction *)     f (-3) (* application *)
+ f 3 (* is +(f 3) *)     - f 3    (* is -(f 3) *)
```