# Unit 1 Introduction

**Unit Outcomes.** Here you will learn

- `why it is useful to study PL concepts`
- `what a PL is for and how different PL paradigms fulfil this task`
- `what makes a good PL and why there is no perfect PL`





2018-01-21

CS2130 Programming Language Concepts, 2017/2018

└ Unit 1 Introduction

Unit 1 Introduction

Unit Outcomes. Here you will learn
- why it is useful to study PL concepts
- what a PL is for and how different PL paradigms fulfil this task
- what makes a good PL and why there is no perfect PL

1 Why study programming language theory?
2 What is a PL for?
3 Methods of execution
4 PL paradigms
Imperative PLs
Functional PLs
Logic PLs
Procedural vs OO PLs
Python script example
Ada procedure example
5 A perfect PL?

# Why study programming language theory?

You live a new life for every new language you speak.
(a Czech saying, loosely translated)



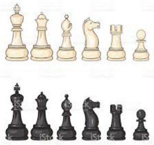
imperative PLs
functional PLs
logic PLs
OO





- improved background for choosing an appropriate PL
- increased ability to learn new PLs
- increased capacity to express programming ideas
- better understanding of the significance of implementation
- overall advancement of computing and an improvement in the quality of software
- increased ability to design good quality new languages

# What is a PL for? (High level)

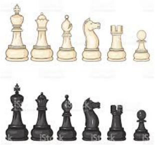






- teach the computer do something for us, typically:
  - *interact with us*
    (eg play games, help us in our tasks)
  - *interact with other computers*
    (eg serve web pages)
  - *manipulate some data*
    (always needed, eg game state, DB $\longrightarrow$ HTML)
- these are often very high-level tasks
- hard to express in computer's terms,
  ie numbers, numbers, . . . nothing but numbers
- a PL helps the programmer: *simulate a more sophisticated machine*

# What is a PL for? (High level)




Real programmers code in binary.







- teach the computer do something for us, typically:
  - *interact with us*
    (eg play games, help us in our tasks)
  - *interact with other computers*
    (eg serve web pages)
  - *manipulate some data*
    (always needed, eg game state, DB $\longrightarrow$ HTML)
- these are often very high-level tasks
- hard to express in computer's terms,
  ie numbers, numbers, . . . nothing but numbers
- a PL helps the programmer: *simulate a more sophisticated machine*

# What is a PL for? (Low level)

- can we use a precise version of a natural language as a PL?





- typically a PL allows us to:
  - represent *structured* data in computer memory
  - perform *basic* ops on the data
  - access *IO* devices, react to events
  - carry out specified steps in some order, depending on the data





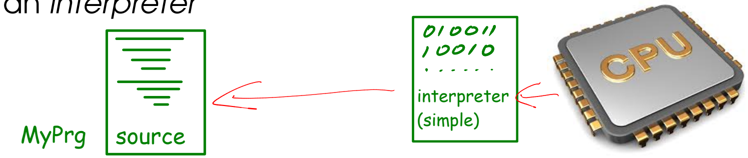
What is a PL for? (Low level)
- can we use a precise version of a natural language as a PL?
- typically a PL allows us to:
  - represent *structured* in computer memory
  - perform *basic* on the data
  - access *IO* , react to
  - carry out specified in some order, depending on the data

- can we use a precise version of a natural language as a PL?

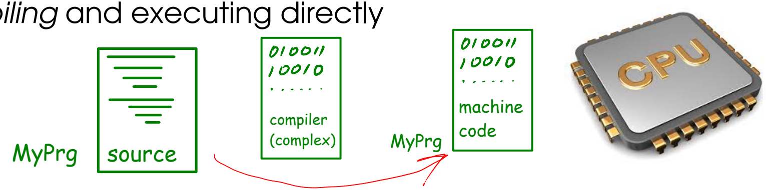
  eg *Please, play chess with me! In chess you move like this. . .*
- yes but. . . how to execute it efficiently?
- a PL cannot go too far from the underlying computer architecture
- a PL must stick to fairly *simple* and *precise* terms

# Methods of execution

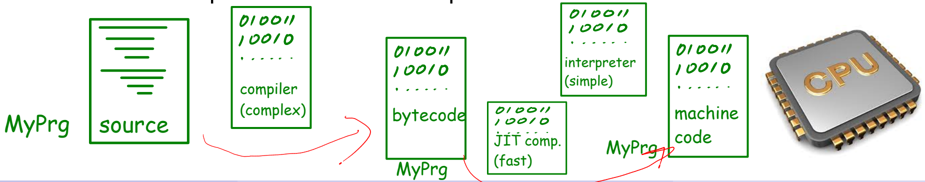
- using an *interpreter*




- *compiling* and executing directly

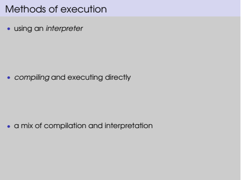



- a mix of compilation and interpretation



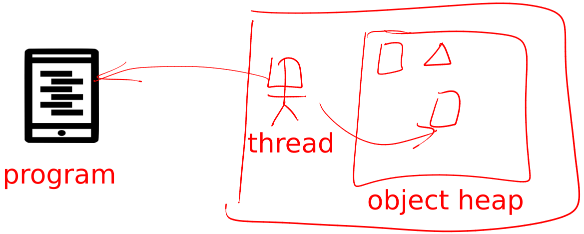





- using an *interpreter*
  - interpreter = program that simulates the PL's machine
  - interpreter reads and executes programs at the same time
- *compiling* and executing directly
  - program → equivalent program in native machine code
  - runs the program much faster then an interpreter
  - difficult to port between different types of computer
- a mix of compilation and interpretation
  - eg Java
    - compiling to a very low-level language — *byte code*
    - byte code is portable, interpreted fairly efficiently
  - some interpreters compile — JIT (just-in-time) compilation
  - some compilers interpret: interpreter included in executable
  - many compilers compromise: generate native machine code + large agent that helps or directs the execution

- in essence: *tell the computer what to do step-by-step*








PL paradigms
Imperative PLs
- in essence: *tell the computer what to do step-by-step*

- the simulated machine: one or more "agents" called *threads* + *data store*
- threads read our instructions and carry them out one-by-one

## Essence of Functional Programming (FP)

program = expression = data        computation = prg. evolution

$$
\begin{aligned}
a(x) &:= (x-1)*(x+1) && \text{definition of } a\\
a(3) &= (3-1)*(3+1) = 2*4 = 8\\[1ex]
a(x) &= x^2-1 && \text{optimisation of } a\text{, using rules}\\
a(3) &= 3^2-1 = 9-1 = 8\\[1ex]
add1(xs) &:= map\ (+1)\ xs\\
add1([1,2,3]) &= map\ (+1)[1,2,3] = [2,3,4]\\[1ex]
p(x,s) &:= \text{replicate } x \text{ (print } s)
\end{aligned}
$$

algebraic rules
eg x-x=0
(x-1)(x+1)
=x^2-1





PL paradigms
Functional PLs
Essence of Functional Programming (FP)
program = expression = data computation = prg. evolution
$a(x) := (x-1)*(x+1)$ definition of $a$
$a(3) = (3-1)*(3+1) = 2*4 = 8$
$a(x) = x^2-1$ optimisation of $a$, using rules
$a(3) = 3^2-1 = 9-1 = 8$
$add1(xs) = map\ (+1)\ xs$
$add1([1,2,3]) = map\ (+1)[1,2,3] = [2,3,4]$
$p(x,s) :=$ replicate $x$ (print $s$)

- the simulated machine evaluates expressions
  - in functional programming: program = expression = data
  - computation = simplifying the program using simple rules
  - like a symbolic calculator...
  - when program cannot be simplified, that is the result
- expressions define mathematical functions
  - functions get parameters and return a value
  - pure: no other effects (ie no *side-effects*)
  - the result can be an imperative program with side-effects:

program = expression = data    computation = prg. evolution

$$add1(xs) = \; map\ (+1)\ xs$$
$$add1([1,2,3]) = \; map\ (+1)[1,2,3] = [2,3,4]$$


$$p(x,s) := \; \text{replicate } x \text{ (print } s\text{)}$$



2018-01-21

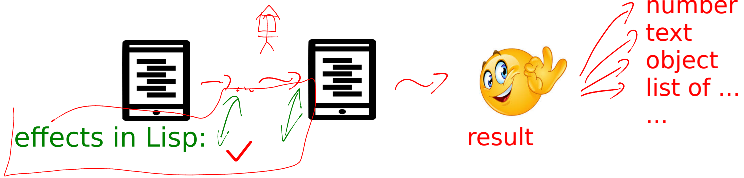
CS2130 Programming Language Concepts, 2017/2018
└PL paradigms
└Functional PLs
└PL paradigmsFunctional PLs

PL paradigms
Functional PLs
program = expression = data computation = prg. evolution
add1(xs) = map (+1) xs
add1([1,2,3]) = map (+1)[1,2,3] = [2,3,4]
p(x,s) := replicate x (print s)

program = expression = data    computation = prg. evolution

$$add1(xs) = \quad map\ (+1)\ xs$$

$$add1([1,2,3]) = \quad map\ (+1)[1,2,3] = [2,3,4]$$




$$p(x,s) := \quad \text{replicate } x \text{ (print } s\text{)}$$

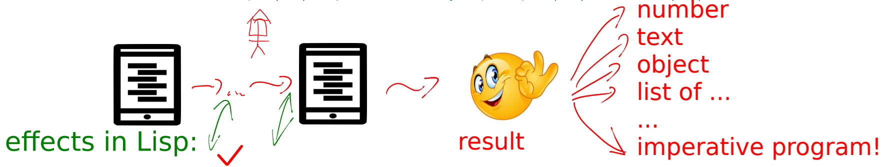




PL paradigms
Functional PLs
program = expression = data computation = prg. evolution
add1(xs) = map (+1) xs
add1([1,2,3]) = map (+1)[1,2,3] = [2,3,4]
p(x,s) := replicate x (print s)

program = expression = data  computation = prg. evolution

$$add1(xs) = map\ (+1)\ xs$$
$$add1([1,2,3]) = map\ (+1)[1,2,3] = [2,3,4]$$




$$p(x,s) := \text{replicate } x \text{ (print } s\text{)}$$




2018-01-21

CS2130 Programming Language Concepts, 2017/2018
└PL paradigms
└Functional PLs
└PL paradigmsFunctional PLs

PL paradigms
Functional PLs
program = expression = data computation = prg. evolution
add1(xs) = map (+1) xs
add1([1,2,3]) = map (+1)[1,2,3] = [2,3,4]
p(x,s) := replicate x (print s)

# Functional PLs — Lisp mini introduction

```
(format nil "hello ~d" 123)  ;; returns "hello 123"
;; in Java syntax: format(null, "hello  d", 123);

;; an expression returns a value and also can have side effects:
(print "hello")  ;; prints "hello" to console, then returns "hello"

;; declare, assign and use a variable:
(setf name "Alice")  ;; in Java: String name; name = "Alice";

;; revisit print: returns a value in addition to printing:
(setf name2 (print "Bob"))

;; operators are just functions:
(setf n (+ 1 2))  ;; in Java: int n; n = 1 + 2;

;; "if" and comparison are also functions:
(if (not (string= name "")) name "noname")

;; sequencing multiple expressions:
(progn (print "hello:") (print name))

;; a program can be treated as data:
(setf myprogram '(print (+ 1 m)))  ;; the tick is important
myprogram  ;; returns (PRINT (+ 1 M))
;; (eval myprogram) ;; throws error: "EVAL: variable M has no value"

(setf m 1)
(eval myprogram)   ;; prints 2
```

# Functional PLs — Lisp example

```lisp
;; pure functions:
(defun circleLines (size)
  (loop for i from 1 to size collect (circleLine size i)))

(defun circleLine (size i)
  (concat-strings
   (loop for jj from 1 to (* 2 size)
     collect (circleChar size i jj))))

(defun circleChar (size i jj)
  (let
    ((j (truncate jj 2)))
    (if (shouldPaint size i j) "*" " ")))

(defun shouldPaint (size i j) ... ) ;; shortened

;; An impure (imperative) function to print a list of lines:
(defun write-lines (lines)
  (loop for line in lines do (write-line line)))

(write-lines (circleLines (parse-integer (first *args*))))
```



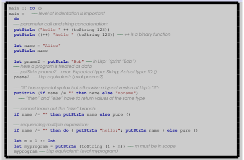


Functional PLs — Lisp example

```
main :: IO ()
main =    -- level of indentation is important
  do
  -- parameter call and string concatenation:
  putStrLn ("hello " ++ (toString 123))
  putStrLn ((++) "hello " (toString 123)) -- ++ is a binary function

  let name = "Alice"
  putStrLn name

  let pname2 = putStrLn "Bob" -- in Lisp: '(print "Bob")
  -- here a program is treated as data
  -- putStrLn pname2 - error: Expected type: String; Actual type: IO ()
  pname2 -- Lisp equivalent: (eval pname2)

  -- "if" has a special syntax but otherwise a typed version of Lisp's "if":
  putStrLn (if name /= "" then name else "noname")
    -- "then" and "else" have to return values of the same type

  -- cannot leave out the "else" branch:
  if name /= "" then putStrLn name else pure ()

  -- sequencing multiple expressions:
  if name /= "" then do { putStrLn "hello:"; putStrLn name } else pure ()

  let m = 1 :: Int
  let myprogram = putStrLn (toString (1 + m)) -- m must be in scope
  myprogram -- Lisp equivalent: (eval myprogram)
```







# Functional PLs — Haskell example

```
main :: IO () — type of imperative programs
main =
  do — introduces a sequence of steps
  [sizeS] <- getArgs
  sequence_ (map putStrLn (circleLines (read sizeS)))

circleLines :: Integer -> [String] — pure function type
circleLines size =
  map (circleLine size) [1..size] — a range list using enumeration
      — partial application - one parameter omitted

circleLine :: Integer -> Integer -> String
circleLine size i =
  map (circleChar size i) [1..(size*2)]

circleChar :: Integer -> Integer -> Integer -> Char
circleChar size i jj =
  if shouldPaint then '*' else ' '
  where
  shouldPaint = ... — definition omitted
```

(all types can be omitted, inferred by compiler)

# Functional PLs — pros and cons

- Sometimes people find FP hard to learn
  - unfamiliar, very different set of basic tricks
  - complex tricks make code hard to read
- Why use FP?
  - short programs with clear meaning
  - compiler can make many checks and optimisations
  - fast development, less debugging needed
  - good reusability
  - easier programming of multi-threaded computation





PL paradigms
Functional PLs — pros and cons
- Sometimes people find FP hard to learn
  - unfamiliar, very different set of basic tricks
  - complex tricks make code hard to read
- Why use FP?
  - short programs with clear meaning
  - compiler can make many checks and optimisations
  - fast development, less debugging needed
  - good reusability
  - easier programming of multi-threaded computation

## Essence of Logic Programming

program = knowledge = logical rules for facts about data
computing = checking facts, filling gaps

```
range([Max], Max, Max). % a range with only one number
range([Min | Rest], Min, Max) :-
      % prolog uses the (head | tail) syntax for lists
  MinPlusOne is Min + 1,
  range(Rest, MinPlusOne, Max). % recursion - one less
```

- `range([1,2,3],1,3).` . . . *(computation)*. . . YES
- `range(X,1,4).` . . . *(computation)*. . . YES if `X = [1,2,3,4]`
- `range([1,2,3],X,Y).` . . . *(computation)*. . . YES if `X = 1, Y = 3`
- `range([1,2,3],2,X).` . . . *(computation)*. . . NO





- the simulated machine: agent understands knowledge and makes deductions from them
- agent can be asked questions, eg `range([1,2],1,2).`?
- agent looks up the rules and says YES/NO
- logic PLs are good for certain AI applications

- Prolog syntax: variable names start with a capital letter
- here, `range` takes 3 parameters and gives a Boolean
- `range(X,Y,Z)` represents some fact about X,Y,Z

## Essence of Logic Programming

program = knowledge = logical rules for facts about data
computing = checking facts, filling gaps

```
range([Max], Max, Max). % a range with only one number
range([Min | Rest], Min, Max) :-
      % prolog uses the (head | tail) syntax for lists
  MinPlusOne is Min + 1,
  range(Rest, MinPlusOne, Max). % recursion - one less
```


- `range([1,2,3],1,3).` ...*(computation)*...YES
- `range(X,1,4).` ...*(computation)*...YES if `X = [1,2,3,4]`
- `range([1,2,3],X,Y).` ...*(computation)*...YES if `X = 1, Y = 3`
- `range([1,2,3],2,X).` ...*(computation)*...NO



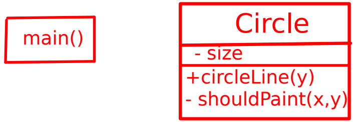


PL paradigms
Logic PLs
Essence of Logic Programming
program = knowledge = logical rules for facts about data
computing = checking facts, filling gaps
range([1,2,3],1,3). ...(computation)...YES
range(X,1,4). ...(computation)...YES if X = [1,2,3,4]
range([1,2,3],X,Y). ...(computation)...YES if X = 1, Y = 3
range([1,2,3],2,X). ...(computation)...NO

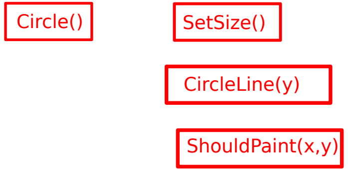
- the simulated machine: agent understands knowledge and makes deductions from them
- agent can be asked questions, eg `range([1,2],1,2).`?
- agent looks up the rules and says YES/NO
- logic PLs are good for certain AI applications

- Prolog syntax: variable names start with a capital letter
- here, `range` takes 3 parameters and gives a Boolean
- `range(X,Y,Z)` represents some fact about X,Y,Z

- OO




- Procedural








PL paradigms
Procedural vs OO PLs
• OO
• Procedural

- imperative languages differ in ways they structure code:
- *object-oriented* PLs (eg Java, C++, Python)
  - strongly associate definitions of actions with definitions of data structures on which the actions operate
  - better support OO design principles and abstractions
- *procedural* PLs (eg Ada, C)
  - action and data definitions are associated only loosely
  - can implement OO designs but require more work from the programmer

# Python script example

```
class Circle:
    def __init__(self, size):
        self.size = size
    def circleLine(self, i):
        result = ""
        for jj in range(1,2*self.size+1):
            if self.shouldPaint(i,jj):
                result += "*"
            else:
                result += " "
        return result
    def shouldPaint(self,i,jj):
        def shouldPaintS(s,i,j): # local function
            return abs (i**2 + j**2 - s**2) <= s+1
        return shouldPaintS(self.size,i,jj/2)

def main():
    size = int(sys.argv[1])
    c = Circle(size)
    for i in range(1,size+1):
        print c.circleLine(i)

main() # interpreter, now that you know what main is, execute it!
```

constructor

methods

standalone method

top-level instruction





Python script example

# Ada procedure example

procedure ~ method

sub-methods

```
procedure Circle is
  size : Integer;

  procedure SetSize is
    last : Integer;
  begin
    Get(Argument(1),size,last); -- assigns value to size and last
  end SetSize;

  function ShouldPaint(i,jj : Integer) return Boolean; -- declaration

  function CircleLine(i : Integer) return String is
    result : String(1..(2*size)); -- array, memory allocated by size
  begin
    ... -- function body omitted; it uses ShouldPaint
    return result;
  end CircleLine;

  function ShouldPaint(i,jj : Integer) return Boolean is
  begin ... end ShouldPaint; -- function body omitted

begin
  SetSize;
  for i in 1..size loop
    Put_Line(CircleLine(i));
  end loop;
end Circle;
```

# A perfect PL? — what attributes

- *expressive*
  (tagline: can describe many algorithms)
- *concise*
  (tagline: no boring code, please!)
- *safe*
  (tagline: errors prevented or found asap)
- *easy to read*
  (anti-tagline: what on earth is this supposed to do? ...)
- *executes efficiently*
  (anti-tagline: it wants 2000GB RAM?!)
- *supporting reuse of code via abstraction*
  (much better than cut and paste!)
- *easy to learn*
  (tagline: a 5000 page introduction?!...)
- *having good standard libraries* — influences all of the above





A perfect PL? — what attributes
- *expressive*
  (tagline: can describe many algorithms)
- *concise*
  (tagline: no boring code, please!)
- *safe*
  (tagline: errors prevented or found asap)
- *easy to read*
  (anti-tagline: what on earth is this supposed to do? ...)
- *executes efficiently*
  (anti-tagline: it wants 2000GB RAM?!)
- *supporting reuse of code via abstraction*
  (much better than cut and paste!)
- *easy to learn*
  (tagline: a 5000 page introduction?!...)
- *having good standard libraries* — influences all of the above

# A perfect PL? — evaluate your favourite PL

use scale (rubbish=) 0 . . . 5 (=sublime)





A perfect PL? — evaluate your favourite PL
use scale (rubbish=) 0 . . . 5 (=sublime)

# A perfect PL? — impossible. . .

- *Some of the criteria are contradicting each other.*

```
map (+1) xs
```

```
for (i=0; i < xs.length; i++) { xs[i] = xs[i]+1; }
```

- *Many of the criteria are context dependent, eg:*
  - AI
  - OS
  - Safety-critical
  - Mobile code

"See? Perfect!"





A perfect PL? — impossible. . .
Some of the criteria are contradicting each other.
map (+1) xs
for (i=0; i < xs.length; i++) { xs[i] = xs[i]+1; }
Many of the criteria are context dependent, eg:
AI
OS
Safety-critical
Mobile code

- *Some of the criteria are contradicting each other.*
  - eg concise + expressive $\Longrightarrow$ not super efficient
  - PL designers seek compromises
- *Many of the criteria are context dependent.*
  - eg expressiveness depends on what we want to express
  - hundreds of specialised PLs exist
  - also the main PLs are often tuned for some context, eg:
    - Java: Web and Internet programming
    - C: operating system programming
    - Ada: safety-critical applications
    - Lisp, Prolog: artificial intelligence

# Learning Outcomes

**Learning Outcomes.** You should now be able to

- `explain why it is useful to study PL concepts, giving examples of useful concepts`
- `list and discuss the most important attributes of good PLs`
- `explain the essence of imperative, functional and logical programming paradigms`
- `execute given Java, Ada, Python, Lisp, Haskell programs`
- `explain the differences between interpreted and compiled languages`





Learning Outcomes

**Learning Outcomes.** You should now be able to

- explain why it is useful to study PL concepts, giving examples of useful concepts
- list and discuss the most important attributes of good PLs
- explain the essence of imperative, functional and logical programming paradigms
- execute given Java, Ada, Python, Lisp, Haskell programs
- explain the differences between interpreted and compiled languages