1 Introduction

2 Basic concepts of programming languages, using Scheme

3 More advanced Scheme concepts

4 Object-oriented programming
Main coordinates

1. **email** matteo.pradella@polimi.it
2. **office** DEIB, Politecnico di Milano
   via Golgi 42, building 22, floor 3, room 322
3. **phone** 3495
4. **web-page** http://home.deib.polimi.it/pradella/PL.html
5. **twitter** @bzoto, tag #corsopl
Motivation and preliminaries

1. Programming languages are tools for writing software
2. I.e. tools for talking to the machine
3. But not only to the machine: often code is created also to be read by human beings
4. (Debugging, pair programming, modifications/extensions)
Motivation and preliminaries (cont.)

1. several levels of abstraction in a computer: hw, operating system, network, applications, daemons, virtual machines, frameworks, web, clouds, ...
2. no "holy grail" language
3. different languages are to be used for programming at different levels
4. close to human: abstract, hard to "control"
5. close to machine: too many details, hard to understand what is going on
6. various (often conflicting) aspects: expressiveness and conciseness; ease of use; ease to control; efficiency of compiled code ...
why is a language successful?
sometimes right tool at the right time; often hype, good marketing, good tools, luck, who knows...
e.g. often is better a so-so language with great compilers, than a very nice language with a partial implementation
(thanks especially to Internet and open source software:) many new languages, strong evolution in recent years
but often the concepts are those introduced 30+ years ago!
Recent, competitive technologies are based on new languages, especially w.r.t. the Web.

- e.g. Ruby on Rails, Node.js
- example of Twitter: Ruby on Rails, core rewritten in Scala
- more and more new technologies and language-based frameworks emerge (or re-emerge, think about Objective-C...)

To adapt, or to "disrupt", we need to know and understand:

- not particular languages, but their **main principles and concepts**
An incomplete timeline of PLs

1. 1957 Fortran (Formula Translator)
2. 1958 LISP (LISt Processor)
3. 1959 COBOL (Common Business Oriented Language)
4. 1960 ALGOL 60 (Algorithmic Language)
5. 1964 BASIC (Beginner’s All-purpose Symbolic Instruction Code)
6. 1967 Simula (first object-oriented lang.)
7. 1970 Pascal, Forth
8. 1972 C, Prolog, Smalltalk
9. 1975 Scheme (Lisp + Algol)
10. 1978 ML (Meta-Language)
11. 1980 Ada
An incomplete timeline of PLs (cont.)

1. 1983 C++, Objective-C
2. 1984 Common Lisp (Lisp + OO)
3. 1986 Erlang
4. 1987 Perl
5. 1990 Haskell
6. 1991 Python
7. 1995 Java, JavaScript, Ruby, PHP
8. 2001 C#
9. 2002 F#
10. 2003 Scala
11. 2007 Clojure
12. 2009 Go; ’11 Dart, ’12 Rust, ’14 Swift ...
A measure of language popularity

RedMonk Q116 Programming Language Rankings

Popularity Rank on GitHub (by # of Projects)

Popularity Rank on Stack Overflow (by # of Tags)
Pre-(and post-)requisites

1. Good knowledge of procedural and object-oriented programming (I assume with C and Java, respectively)
2. Formal languages and compilers
3. Exam: written exercises, small programs, translations from different paradigms. Emphasis on concepts and elegance of the chosen approach; no code obfuscation contest!
Map of used languages

1. **Scheme** (mainly functional), for basics, memory management, introduction to functional programming and object orientation, meta-stuff

2. **Haskell** (functional), for static type systems and algebraic data types, functional purity, parallelism

3. **Prolog**, for logic programming

4. "Academic" languages (but with reference with more mainstream ones): they are simpler, more orthogonal and with less "cruft"

5. We need to understand the **concepts** and to be able to adapt and learn to new languages with little effort. (It is useless to focus on one particular, temporarily successful language.)
Map of concepts

PL Concepts

Procedural programming (S)
Evaluation
- call by value (S)(H*)
- call by reference (S)
- call by name (H)
- call by need (S*)
- memoization (S*)

Continuations (S)

Exceptions (S)(H)

Object-oriented programming (S)

Functional programming
- 1st class functions (S)
- (tail) recursion (S)(H)
- purity (H)
- monads (H)(S*)

Logic programming (P)
- logic concepts
- procedural semantics
- cut + control (S*)
- quoting + macros (S)

Meta programming
- DSL (S)(P)

Concurrency + Parallelism (H)

Type discipline
- dynamic (S)(P)
- parametric polymorphism
- Abstract Data Types
- ad-hoc polymorphism
- type inference

Matteo Pradella
Principles of Programming Languages (I)
April 2016 12 / 104
Scheme: Why?

1. Scheme is a language of the ancient and glorious Lisp Family
2. *[Scheme is intended to] allow researchers to use the language to explore the design, implementation, and semantics of programming languages.* (From the R6RS standard)
3. It is unique because it has extremely simple and flexible syntax and semantics, very few basic ideas
4. Good to understand and experiment new concepts/constructs
5. We will build new constructs and an OO language with it
Main (and free) material:

We will use the Racket dialect of Scheme, which has a very good implementation and environment: http://racket-lang.org/

The last standard is "The Revised Report on the Algorithmic Language Scheme" (aka R7RS).


#lang directive at the beginning of the file to choose the Scheme dialect that we are using (in our case #lang racket)
The obligatory quotes on Lisp

1. Anyone could learn Lisp in one day, except that if they already knew Fortran, it would take three days. – Marvin Minsky

2. If you give someone Fortran, he has Fortran. If you give someone Lisp, he has any language he pleases. – Guy Steele

3. A Lisp programmer knows the value of everything, but the cost of nothing. – Alan Perlis
The typical procedure call \( f(x, y); \) (C/Java) is written like this: \((f \times y)\)

No special syntax for expressions, no infix operators, no precedence rules. E.g.

\[
x == y + 3 \times x + z;
\]

is written

\[
(= x (+ y (* 3 x) z))
\]

Such expressions are called **s-expressions**
Basic types

1. Booleans: #t, #f
2. Numbers: 132897132989731289713, 1.23e+33, 23/169, 12+4i
3. Characters: #\a, #\Z
4. Symbols: a-symbol, another-symbol?, indeed!
5. Vectors: #(1 2 3 4)
6. Strings: "this is a string"
7. Pairs and Lists: (1 2 #\a), (a . b), () (we’ll see later)
Expressions

1. Scheme is mainly a **functional** language, so every program is an **expression**, and computation is based on evaluating expressions (no statements).

2. Evaluation of an expression produces a **value** (if it terminates).

3. Evaluation of an expression \((e_1 \ e_2 \ e_3 \ldots)\) is based on the evaluation of \(e_1\), which identifies an operation \(f\) (e.g. is the name of a procedure). The other sub-expressions (i.e. \(e_i, \ i > 1\)) are evaluated, and their values are passed to \(f\).

4. The evaluation order of \(e_i, \ i \geq 1\) is **unspecified**, e.g. \(e_4\) could be evaluated before \(e_2\).
lambdas are unnamed procedures:

(λ( x y) ; this is a comment
  (+ (* x x) (* y y)))

example usage ((λ( x y) (+ (* x x) (* y y))) 2 3) ⇒ 13
i.e. (λ( x, y) := x^2 + y^2)(2, 3)

Lambdas are called blocks in Smalltalk, Ruby, Objective-C, and are present
in many languages (e.g. in C++11, Java 8).

Procedures are values in Scheme, hence are first class objects.

lambda-calculus introduced by Alonzo Church in the ’30s, theory of
computable functions based on recursion (i.e. recursive functions)
Variables and binding

1. \texttt{let} is used for binding variables:

\begin{verbatim}
(let ((x 2) ; in Scheme
     (y 3))
  ...)
\end{verbatim}

\begin{verbatim}
{ int x = 2, y = 3; // in C
  ... }
\end{verbatim}

2. Scoping rules are \textbf{static} (or lexical): traditional/old Lisps are dynamic (e.g. Emacs Lisp). Scheme was the first Lisp taking static scoping from Algol.
Consider this code:

```scheme
(let ((a 1))
  (let ((f (lambda ()
              (display a))))
    (let ((a 2))
      (f))))
```

1. With static scoping rules, it prints 1; with dynamic scoping rules, 2.

2. A few interpreted languages are still based on dynamic scoping. Some languages can optionally support it (e.g. Common Lisp, Perl). In Perl “my” variables are static, while “local” are dynamic.
let binds variables “in parallel”, e.g.:

(let ((x 1)
   (y 2))
   (let ((x y) ; swap x and y
       (y x))
       ...x))

Evaluates to 2. There is a “sequential” variant called let*. With it, x is bound before y.

if mutual recursion is needed, there are also letrec and letrec*
Scheme, like Lisp and a few other languages (e.g. Prolog), is homoiconic, i.e. there is no distinction between code and data (like machine code in the von Neumann architecture).

This can be cumbersome, e.g. the prefixed full-parenthesizes syntax is not for everyone, but it can be very effective for meta-programming:

As code-is-data, it is very easy to define procedures that build and compose other procedures.

We will consider this aspect later, with many examples.
Not everything is a procedure or a value. E.g. **if** in general does not evaluate all its arguments (if `<condition> <then-part> <else-part>`)

variants: (when `<condition> <then-part>`), (unless `<condition> <else-part>`) 

**if** is a **syntactic form**. In Scheme it is possible to define new syntax through **macros** (we will see them later).

e.g. try to evaluate `+` and **if** at the REPL (read-eval-print-loop)
1. There is a syntax form that is used to prevent evaluation: (quote <expr>)
2. <expr> is left unevaluated.
3. Shorthand notation: 'expr
4. e.g. (quote (1 2 3)) is a list – without the quote, ⇒ error
   (quote (+ 2 3)) is another list – without the quote, ⇒ 5
quasiquote (') and unquote (,) are used for partial evaluation

For example, with shorthand notation:

\[
\text{(quote (1 2 3))} \Rightarrow (1 \ 2 \ 3)
\]

\[
\text{(quasiquote ; (quote (+ 1 1) 3))} \Rightarrow (1 \ 2 \ 3)
\]
procedure **eval** is typical of Lisps, and it is present in many Lisp-inspired/derived languages, e.g. Python, Ruby, JavaScript. . .

in such languages, it has one argument, which is a string containing code to be evaluated

in Scheme, it is just code (e.g. the list \((+ 1 2)\))

it is the "inverse" of quote: \(\texttt{eval } '(+ 1 2 3))\) is 6
If we are writing a block of procedural code, we can use the `begin` construct:

```
(begin
  (op_1 ...)
  (op_2 ...)
  ...
  (op_n ...))
```

Every `op_i` is evaluated in order, and the value of the `begin` block is the value obtained by the last expression.
Definitions

1. Variables created by let are local. To create top-level bindings there is `define`:
   
   ```
   (define <name> <what>)
   ```

2. e.g. 
   
   ```
   (define x 12)
   (define y #(1 2 3))
   ```

3. Note that defining a procedure is no different: 
   
   ```
   (define cube (lambda (x) (* x x x)))
   ```
   ```
   (cube 3) ⇒ 27
   ```

4. `define` can be also used instead of `let` in procedures
Defining procedures

1. There is a shorthand notation for defining procedures, that mimics their usage: `(define (cube x) (* x x x))`

2. `set!` is for assignment:

   ```
   (begin
     (define x 23)
     (set! x 42)
     x)
   ```

   evaluates to 42.

3. NB: in general, procedures with side effects have a trailing bang (!) character.
Lists

1. Lisp traditionally stands for LISt Processor and Scheme takes lists management directly from Lisp.

2. Lists are memorized as concatenated **pairs**, a pair (written \((x.y)\) and also called a **cons** node) consists of two parts:
   1. **car** (i.e. \(x\)) aka *Content of the Address Register*
   2. **cdr** (i.e. \(y\)) aka *Content of the Data Register*

3. A list \((1\ 2\ 3)\) is memorized like this \((1.(2.(3.())))\)

4. \((\)\) is the **empty list** also called **nil** in Lisp.

5. The two procedures **car** and **cdr** are used as accessors.

6. To check if a list contains a value, use **member**: e.g. `(member 2 '(1 2 3))` is `(2 3)`
Lists and procedures

1. Procedures bodies and parameter lists are all plain lists
2. This can be used to implement procedures with a variable number of arguments
3. E.g.
   \[
   \text{(define (x . y) y)} \quad \text{(x 1 2 3)} ;; \Rightarrow '(1 2 3)
   \]
4. \text{apply} can be used to apply a procedure \(p\) to a list of elements
   \[
   \text{(apply + '(1 2 3 4)) ;; \Rightarrow 10}
   \]
5. To build a pair we can use \text{cons}: e.g. \((\text{cons} 1 2)\) is \((1 . 2)\); \((\text{cons} 1 '2))\) is \((1 \ 2)\)
A classical example on lists

1. find the minimum of a list

\[
\begin{align*}
\text{(define (minimum L)} \\
\text{ (let ((x (car L))} \\
\text{ (xs (cdr L)))} \\
\text{ (if (null? xs) ; is xs = ()?} \\
\text{ x ; then return x} \\
\text{ (minimum ; else: recursive call} \\
\text{ (cons} \\
\text{ (if (< x (car xs))} \\
\text{ x} \\
\text{ (car xs))} \\
\text{ (cdr xs)))})}) \\
\text{(minimum '(11 -3 2 3 8 -15 0)) ; => -15}
\end{align*}
\]
A classical example on lists (cont.)

a variant with variable number of arguments:

```
(define (minimum x . rest)
  (if (null? rest) ; is rest = ()?
    x ; then return x
    (apply minimum ; else: recursive call
      (cons
        (if (< x (car rest))
          x
          (car rest))
        (cdr rest))))

(minimum 11 -3 2 3 8 -15 0) ; => -15
```
General loops: the named \texttt{let}

1. Let us start with a "non-idiomatic" example:

$$
\text{(let ((x 0))}
  \text{(let label () ; why an empty list?}
    \text{(when (< x 10)}
      \text{(display x)}
      \text{(newline)}
      \text{(set! x (+ 1 x))}
      \text{(label)())}) ; \text{go-to label}
$$

2. in C or Java:

```c
for (x = 0; x < 10; x++) {
    printf("%d/n", x);
}
```
the strange empty list is used for variables that are used in the loop

indeed, this is the correct, idiomatic way of doing the same thing:

```
(let label ((x 0))
  (when (< x 10)
    (display x)
    (newline)
    (label (+ x 1))))) ; x++
```

of course we can use as many variables as we like

like with `begin`, the value is the one obtained by the last expression
Proper tail recursion

1. Every Scheme implementation is required to be properly tail recursive.

2. A procedure is called tail recursive if its recursive call is "at the tail", i.e. is the last operation performed.

3. E.g. not tail recursive:

   ```scheme
   (define (factorial n)
     (if (= n 0)
       1
       (* n (factorial (- n 1)))))
   ```

4. E.g. tail recursive:

   ```scheme
   (define (fact x)
     (define (fact-tail x accum) ; local proc
       (if (= x 0) accum
         (fact-tail (- x 1) (* x accum))))
     (fact-tail x 1))
   ```
Tail recursive procedures can be optimized to avoid stack consumption indeed, the previous tail call is translated in the following low-level code:

```
(define (fact-low-level n)
  (define x n)
  (define accum 1)
  (let loop () ;; see this as the "loop" label
   (if (= x 0)
     accum
     (begin
       (set! accum (* x accum))
       (set! x (- x 1))
       (loop))))) ;; jump to "loop"
```

Matteo Pradella
Principles of Programming Languages (I)
April 2016
of course, a more idiomatic way of writing it is the following:

```
(define (fact-low-level-idiomatic n)
  (let loop ((x n)
             (accum 1))
    (if (= x 0)
      accum
      (loop (- x 1)(* x accum)))))
```

but note that this looks like a tail call...

(In reality, the named let is translated into a local recursive function. If tail recursive, when compiled it becomes a simple jump.)
Nothing much to say, besides the syntax:

```scheme
(for-each (lambda (x)
            (display x)(newline))
  '(this is it))
```
for-each for vectors

1. es:

```
(vector-for-each (lambda (x)
    (display x)(newline))
  #(this is it))
```

2. here is the definition:

```
(define (vector-for-each body vect)
  (let ((max (- (vector-length vect) 1)))
    (let loop ((i 0))
      (body (vector-ref vect i)) ; vect[i] in C
      (when (< i max)
        (loop (+ i 1))))))
```
A predicate is a procedure that returns a Boolean. Its name usually ends with ? (e.g. we already saw null?)

= is used only for numbers

there are eq?, eqv?, and equal?

very roughly:

1. eq? tests if two objects are the same (good for symbols)
   - (eq? 'casa 'casa), but not (eq? "casa" "casa"), (eq? 2 2) is unspecified

2. eqv? like eq?, but checks also numbers

3. equal? predicate is #t iff the (possibly infinite) unfoldings of its arguments into regular trees are equal as ordered trees.
   - (equal? (make-vector 5 'a)(make-vector 5 'a)) is true
case and cond

1. case:
   
   ```lisp
   (case (car '(c d))
      ((a e i o u) 'vowel)
      ((w y) 'semivowel)
      (else 'consonant)); => consonant
   ```

2. cond:
   
   ```lisp
   (cond ((> 3 3) 'greater)
         ((< 3 3) 'less)
         (else 'equal)); => equal
   ```

3. Note: they are all symbols; neither strings, nor characters

4. the predicate used in case is eqv?
Variables and object implicitly refer to locations or sequence of locations in memory (usually the heap)

Scheme is garbage collected: every object has unlimited extent - memory used by objects that are no longer reachable is reclaimed by the GC

Constants reside in read-only memory (i.e. regions of the heap explicitly marked to prevent modifications), therefore literal constants (e.g. the vector #(1 2 3)) are immutable

If you need e.g. a mutable vector, use the constructor (vector 1 2 3)

Mutation, when possible, is achieved through "bang procedures", e.g. (vector-set! v 0 "moose")
Example on literal constants and mutability

with standard constructors, e.g. vectors are mutable

(define (f)
  (let ((x (vector 1 2 3)))
    x))

(define v (f))
(vector-set! v 0 10)
(display v)(newline) ; => #(10 2 3)
literal constants should be immutable:

(define (g)
  (let ((x #(1 2 3))
        x))

(display (g))(newline) ; => #(1 2 3)
(vector-set! (g) 0 10) ; => error!

in Racket, lists are immutable (so no set-car!, set-cdr!) - this is different from most Scheme implementations (but it is getting more common)

There is also a mutable pair datatype, with mcons, set-mcar!, set-mcdr!
Evaluation strategy: **call by object sharing** (like in Java): objects are allocated on the heap and references to them are passed **by value**.

```scheme
(define (test-setting-local d)
  (set! d "Local") ; setting the local d
  (display d)(newline))

(define ob "Global")
(test-setting-local ob) ;; => Local
(display ob) ;; => Global
```
It is also often called **call by value**, because objects are evaluated before the call, and such values are **copied** into the activation record.

The copied value is not the object itself, which remains in the heap, but a **reference** to the object.

This means that, if the object is mutable, the procedure may exhibit **side effects** on it.

```scheme
(define (set-my-mutable d)
  (vector-set! d 1 "done")
  (display d))

(define ob1 (vector 1 2 3)) ;; i.e. #(1 2 3)
(set-my-mutable ob1) ;; => #(1 done 3)
(display ob1) ;; => #(1 done 3)
```
Introducing new types: structs

1. It is possible to define new types, through `struct`.
2. The main idea is like `struct` in C, with some differences.
3. e.g.
   
   ```
   (struct being ( 
       name ; name is immutable 
       (age #:mutable) ; flag for mutability 
   ))
   ```

4. a number of related procedures are automatically created, e.g. the constructor `being` and a predicate to check if an object is of this type: `being?` in this case.
also accessors (and setters for mutable fields) are created

e.g., we can define the following procedure:

```
(define (being-show x)
  (display (being-name x))
  (display " (")
  (display (being-age x))
  (display ")")
)

(define (say-hello x)
  (if (being? x) ;; check if it is a being
    (begin
      (being-show x)
      (display " : my regards. ")
      (newline))
    (error " not a being " x)))
```


1 example usage:

```scheme
(define james (being "James" 58))
(say-hello james)
;; => James (58): my regards.
(set-being-age! james 60); a setter
(say-hello james)
;; => James (60): my regards.
```

2 clearly it is not possible to change its name
structs can inherit

```
(struct may-being being ; being is the father
 ((alive? #:mutable)) ; to be or not to be
)
```

this being can be killed:

```
(define (kill! x)
  (if (may-being? x)
      (set-may-being-alive?! x #f)
      (error "not a may-being" x)))
```
dead being are usually untalkative:

(define (try-to-say-hello x)
  (if (and
        (may-being? x)
        (not (may-being-alive? x)))
     (begin
       (display "I hear only silence."
       (newline)))
     (say-hello x))))
now we create:

```
(define john (may-being "John" 77 #t))
(say-hello john)
   ; => John (77): my regards.
```

note that John is also a being

and destroy:

```
(kill! john)
(try-to-say-hello john)
   ; => I hear only silence.
```
The main difference is in *methods vs procedures*:

- procedures are *external*, so with inheritance we cannot redefine/override them.

Still, a **may-being** behaves like a **being**

- but we had to define a new procedure (i.e. **try-to-say-hello**), to cover the role of **say-hello** for a **may-being**

- **structs** are called **records** in the standard.
1. A **closure** is a function together with a referencing environment for the non-local variables of that function.

2. i.e. a function object that "closes" over its visible variables.

3. e.g.

   ```scheme
   (define (make-adder n)
     (lambda (x)
       (+ x n)))
   ```

4. It returns an object that maintains its local value of n.

   ```scheme
   (define add5 (make-adder 5))
   (define sub5 (make-adder -5))
   (= (add5 5) (sub5 15)) ; => #t
   ```
Here is a simple application, a closure can be used as an *iterator*:

```scheme
(define (iter-vector vec)
  (let ((cur 0)
        (top (vector-length vec)))
    (lambda ()
      (if (= cur top)
          '<<end>>
          (let ((v (vector-ref vec cur)))
            (set! cur (+ cur 1))
            v))))))

(define i (iter-vector #(1 2)))
(i) ; => 1
(i) ; => 2
(i) ; => '<<end>>
```
An interlude on some classical higher order functions

1. remember the famous map/reduce framework introduced by Google
2. the following operations are supported also by many other languages, e.g. Python and Ruby
3. **map**: \( \text{map}(f, (e_1, e_2, \ldots, e_n)) = (f(e_1), f(e_2), \ldots, f(e_n)) \)
4. **filter**: \( \text{filter}(p, (e_1, e_2, \ldots, e_n)) = (e_i \mid 1 \leq i \leq n, p(e_i)) \)
5. **folds**: **foldr** and **foldl** (aka **reduce** in Python, **inject** in Ruby, **std::accumulate** in C++)
6. let \( \circ \) be a binary operation

\[
\text{fold}\_\text{left}(\circ, \iota, (e_1, e_2, \ldots, e_n)) = (e_n \circ (e_{n-1} \circ \ldots (e_1 \circ \iota)))
\]
\[
\text{fold}\_\text{right}(\circ, \iota, (e_1, e_2, \ldots, e_n)) = (e_1 \circ (e_2 \circ \ldots (e_n \circ \iota)))
\]
Examples

(map (lambda (x) (+ 1 x)) '(0 1 2))
; => (1 2 3)
(filter (lambda (x) (> x 0)) '(10 -11 0))
; => (10)
(foldl string-append ""
    '("una" " " "bella" " " "giornata"))
; => "giornata bella una"
(foldl cons '() '(1 2 3))
; => (3 2 1)
(foldr cons '() '(1 2 3))
; => (1 2 3)
(foldl * 1 '(1 2 3 4 5 6)) ; i.e. factorial
; => 720
**Example implementation of folds**

foldl is tail recursive, while foldr isn’t

```scheme
(define (fold-left f i L)
  (if (null? L)
      i
      (fold-left f
        (f (car L) i)
        (cdr L)))))

(define (fold-right f i L)
  (if (null? L)
      i
      (f (car L)
        (fold-right f i (cdr L))))))
```
Actually, there is a way of making `foldr` tail rec.

\[
\text{(define (fold-right-tail-f f i L)}
\]
\[
\text{(define (fold-right-tail-h f i L out)}
\]
\[
\text{(if (null? L)}
\]
\[
\text{(out i)}
\]
\[
\text{(fold-right-tail-h f i (cdr L)}
\]
\[
\text{(lambda (x)}
\]
\[
\text{(out (f (car L) x)))})})
\]
\[
\text{(fold-right-tail-h f i L (lambda (x) x)))}
\]

The idea is to save the code to be performed \textit{after} the recursive call in a closure.

Do we gain anything, as far as memory occupation is concerned?
If you are fond of **while** loops, rest assured that it is possible to introduce them in Scheme

E.g. the previous factorial could be written like this:

```
(define (fact-with-while n)
  (let ((x n)
        (accum 1))
    (while (> x 0)
      (set! accum (* x accum))
      (set! x (- x 1)))
    accum))
```

clearly, we cannot define it as a procedure (why?)

but how is it possible to extend the syntax?
Meta-programming through macros: i.e. how to program your compiler

1. Scheme has a very powerful, Turing-complete macro system (unlike that of C)
2. Like in C, macros are expanded at compile-time
3. Macros are defined through `define-syntax` and `syntax-rules`
4. `syntax-rules` are pairs (pattern expansion):
   1. **pattern** is matched by the compiler,
   2. then expanded into **expansion**
Let us start with an example: the while loop

```
(define-syntax while
  (syntax-rules () ; no other needed keywords
    ((_ condition body ...) ; pattern P
      (let loop () ; expansion of P
        (when condition
          (begin
            body ...
            (loop))))))
```

_ in the pattern stands for while, ... is a keyword used to match sequences of items
let* as a recursive macro

1. Note that (let ((x 1)) ...) can be expressed with a lambda form:

2. ((lambda (x) ...) 1)

3. So we could define for instance let* as a recursive macro:

```
(define-syntax my-let*
  (syntax-rules ()
    ;; base (= only one variable)
    ((_ ((var val)) istr ...)
      ((lambda (var) istr ...) val))
    ;; more than one
    ((_ ((var val) . rest) istr ...)
      ((lambda (var)
         (my-let* rest istr ...)) val)))
```
and let as a macro

1. It is also very simple to define let:

   (define-syntax my-let
     (syntax-rules ()
       ((_ ((var expr) ...) body ...) body ...)
       ((lambda (var ...) body ...) expr ...)))

2. In it there is an interesting usage of operator ...

3. The first ... in the pattern is used to match a sequence of pairs (var expr), but in the expansion the first ... gets only the var elements, while the last ... gets only the expr elements
Hygiene

1. Scheme macros are *hygienic*
2. this means that symbols used in their definitions are actually replaced with special symbols not used anywhere else in the program
3. therefore it is impossible to have name clashes when the macro is expanded
4. Note that other macro systems, e.g. that of Common Lisp, are not hygienic, so this aspect must be manually managed
5. on the other hand, sometime we *want* name clashes, so these particular cases can be tricky (we will see an example later)
A continuation is an abstract representation of the control state of a program. In practice, it is a data structure used to represent the state of a running program. The current continuation is the continuation that, from the perspective of running code, would be derived from the current point in a program execution. If the language supports first class functions, it is always possible to refactor code in continuation passing style, where the control is passed explicitly in the form of a continuation. (Hint: we saw an example with the tail-recursive fold-right.)
Scheme, unlike many mainstream languages, natively supports continuations:

- **call-with-current-continuation** (or **call/cc**) accepts a procedure with one argument, to which it passes the current continuation, implemented as a closure.

- There are other languages that support first-class continuations: e.g. Ruby (but not JRuby), C in POSIX with `setcontext`, some implementations of JavaScript.

- A similar mechanism is also present in Python, with **generators** (i.e. `yield`).

- Critics also call them **glorified gotos**: they are powerful but abusing them makes hard to understand the program control.
The argument of call/cc is also called an escape procedure;
the escape procedure can then be called with an argument that becomes the result of \texttt{call/cc}.
This means that the escape procedure abandons its own continuation, and reinstates the continuation of \texttt{call/cc} (see next example)
In practice: we save/restore the call stack (we will talk about the implementation later)
A first example

(+ 3
 (call/cc
  (lambda (exit)
   (for-each (lambda (x)
                (when (negative? x)
                 (exit x)))
    '(54 0 37 -3 245 19))
  10)))

1 here we obtain 0

2 Important: an escape procedure has *unlimited extent*: if stored, it can be called after the continuation has been invoked, also multiple times
call/cc: a simple example

```
(define saved-cont #f) ; place to save k

(define (test-cont)
  (let ((x 0))
    (call/cc
      (lambda (k) ; k contains the continuation
        (set! saved-cont k))) ; here is saved

    ;; this *is* the continuation
    (set! x (+ x 1))
    (display x)
    (newline)))
```
let us try it at the REPL:

\[
\begin{align*}
(test\text{-}cont) &; ; \Rightarrow 1 \\
(saved\text{-}cont) &; ; \Rightarrow 2 \\
(define\ other\text{-}cont\ saved\text{-}cont) \\
(test\text{-}cont) &; ; \Rightarrow 1 \text{ (here we reset saved\text{-}cont)} \\
(other\text{-}cont) &; ; \Rightarrow 3 \text{ (other is still going...)} \\
(saved\text{-}cont) &; ; \Rightarrow 2 \\
\end{align*}
\]

What if I put these instructions in a function and call it?
Implementation of call/cc

1. there are various ways of implementing call/cc
2. we consider here two approaches (there are many variants):
   1. the garbage-collection strategy
   2. the stack strategy
in it, we do not use the stack at all: call frames are allocated on the heap
frames that are not used anymore are reclaimed by the GC
call/cc simply saves the **frame pointer** of the current frame
when we call the continuation, we are just setting the current frame pointer
to the one saved before
(note: the stackless implementation of Python works like this)
Stack strategy

1. in it, we use the stack as usual
2. when call/cc is issued, we create a **continuation object** in the heap by copying the current stack
3. when we call the continuation, we reinstate the saved stack, discarding the current one
4. it is a **zero-overhead** strategy: if we do not use call/cc, we do not pay its cost
5. nonetheless, here call/cc can be quite expensive if used
Another example: a for with break

1. The idea is to introduce a For loop with a break-like statement, like:

```lisp
(For i from 1 to 10
do
  (displayln i)
  (when (= i 5)
    (break)))
```

2. Problem with hygienic macros: we need to be able to access to the parameter containing the escape continuation. This is not so easy with syntax-rules.
A simple solution

(define-syntax For
  (syntax-rules (from to break: do)
    ((_ var from min to max break: break-sym
do body ...)
      (call/cc (lambda (break-sym)
        (let ((inc (if (< min max) + -)))
          (let loop ((var min))
            body ...
              (unless (= var max)
                (loop (inc var 1))))))))))

this can be used e.g. like this:

(For i from 1 to 10 break: get-out
do
  (displayln i)
  (when (= i 5)
    (get-out)))
Exception handling is quite common in programming languages (see e.g. Java, where they are pervasive)

Recent Scheme standards have exception handling; Racket has its own variant

We do not want to cover here the details (there is the reference manual for that, and you already know them well), but just show how to implement a throw / catch exception mechanism using continuations
first we need a stack for installed handlers:

```
(define *handlers* (list))

(define (push-handler proc)
  (set! *handlers* (cons proc *handlers*)))

(define (pop-handler)
  (let ((h (car *handlers*)))
    (set! *handlers* (cdr *handlers*)
     h)))
```
Exceptions: Throw

1 throw: if there is a handler, we pop and call it; otherwise we raise an error

(define (throw x)
  (if (pair? *handlers*)
      ((pop-handler) x)
      (apply error x))))
Exceptions: Catch

(define-syntax catch
  (syntax-rules ()
    ((_ what handler exp1 ...)
      (call/cc (lambda (exit)
        ;; install the handler
        (push-handler
          (lambda (x)
            (if (equal? x what)
              (exit handler)
              (throw x)))
        (let ((res ;; evaluate the body
              (begin exp1 ...)))
          ; ok: discard the handler
          (pop-handler)
          res))))))
An example with throw/catch

```
(define (foo x)
  (display x) (newline)
  (throw "hello"))

(catch "hello"
  (begin ;; this is the handler block
    (display "I caught a throw."
    (newline)  
    #f)
  ;; body
  (display "Before foo ")
  (newline)
  (foo "hi!"
  (display "After foo")
  (display "After foo") ; unreached code
```
(call/cc
  (lambda (exit)
    (push-handler
      (lambda (x)
        (if (equal? x "hello")
          (exit (begin
              (display "I caught a throw.")
              (newline)
              #f))
          (throw x))))
    (let ((res (begin
               (display "Before foo ")
               (newline)
               (foo "hi!"
                (display "After foo")))))
      (pop-handler
       res))))
What is Object Oriented programming?

1. First of all, I assume you already know the main concepts from Java.
2. OO means different things to different people.
3. According to Alan Kay, who introduced the term:
   - **OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things. It can be done in Smalltalk and in Lisp. There are possibly other systems in which this is possible, but I’m not aware of them.**
   - **Actually I made up the term "object-oriented", and I can tell you I did not have C++ in mind.**
4. On the other hand, Stroustrup based C++’s OO model on Simula, not Smalltalk. Interesting comparison with Objective-C.
In recent years, there has been a re-thinking about the basics of OO
we can see that in some recent languages, see e.g. Scala, Go, Rust, and Swift
many provide only some of the classical characteristics of OO, e.g. they are
class-less, sometimes don’t offer explicit inheritance, or are based on different
basic ideas, e.g. interfaces
there has also been a gradual shift toward \textit{functional programming} facilities
and principles (see also Java 8 and C++11)
It is possible to use closures to do some basic OO programming. The main idea is to define a procedure which assumes the role of a class. This procedure, when called, returns a closure that works like an object. It works by implementing information hiding through the "enclosed" values of the closure. Access to the state is though messages to a function that works like a dispatcher.
Closures as objects (1)

```
(define (make-simple-object)
  (let ((my-var 0))
    (define (init)
      (set! my-var 0)
      my-var)
    (define (my-add x)
      (set! my-var (+ my-var x))
      my-var)
    (define (get-my-var)
      my-var))
```
(define (my-display)
  (newline)
  (display "my Var is:"
  (display my-var)
  (newline))

(lambda (message . args) ; dispatcher
 (apply (case message
   ((init) init)
   ((my-add) my-add)
   ((my-display) my-display)
   ((get-my-var) get-my-var)
   (else (error "Unknown Method!")))
 args))))
make-simple-object returns a closure which contains the dispatcher

Example usage:

```
(define a (make-simple-object))
(define b (make-simple-object))
(a 'init) ; => 0
(b 'init) ; => 0
(a 'my-add 3) ; => 3
(a 'my-add 4) ; => 7
(a 'get-my-var) ; => 7
(b 'get-my-var) ; => 0
(a 'my-display) ; => My Var is: 7
```
(define (make-son)
  (let ((parent (make-simple-object))
        (name "an object"))
    (define (init)
      (parent 'init))
    (define (hello)
      "hi!"
    )
    (define (my-display)
      (display "My name is ")
      (display name)
      (display " and")
      (parent 'my-display)))
(lambda (message . args)
  (case message
    ((init)      (apply init args))
    ; overriding:
    ((my-display) (apply my-display args))
    ((hello)     (apply hello args))
    (else (apply parent (cons message args))))))
Example usage:

```
(define c (make-son))
(c 'init)
(c 'my-display) ; => My name is an object and
    ; my Var is: 0
(display (c 'hello)) ; => hi!
```
A prototype-based object system

1. **Self** (1987), a variant of Smalltalk, is the programming language that introduced **prototype-based** object orientation.

2. There are no classes: new objects are obtained by **cloning** and modifying existing objects.

3. Its OO model inspired the one of JavaScript.

4. We will see here how to implement it on top of Scheme, using **hash tables** as the main data structure.
An object is implemented with an hash table

\[
\begin{align*}
\text{(define new-object make-hash)} \\
\text{(define (clone object) )} \\
& \quad \text{(hash-copy object))}
\end{align*}
\]

keys are attribute/method names
just for convenience:

```scheme
(define-syntax !! ;; setter
  (syntax-rules ()
    ((_ object msg new-val)
      (hash-set! object 'msg new-val)))))

(define-syntax ?? ;; reader
  (syntax-rules ()
    ((_ object msg)
      (hash-ref object 'msg)))))

(define-syntax -> ;; send message
  (syntax-rules ()
    ((_ object msg arg ...) ((hash-ref object 'msg) object arg ...))))
```
First, we define an object and its methods

```
(define Pino (new-object))
(!! Pino name "Pino") ;; slot added
(!! Pino hello
   (lambda (self x) ;; method added
      (display (?? self name))
      (display ": hi, ")
      (display (?? x name))
      (display "!")
      (newline)))
```
a couple of other methods:

```scheme
(!! Pino set-name
  (lambda (self x)
    (!! self name x)))
(!! Pino set-name-&-age
  (lambda (self n a)
    (!! self name n)
    (!! self age a)))
```

and a clone:

```scheme
(define Pina (clone Pino))
(!! Pina name "Pina")
```
Using the example

(-> Pino hello Pina) ; Pino: hi, Pina!
(-> Pino set-name "Ugo")
(-> Pina set-name-&-age "Lucia" 25)
(-> Pino hello Pina) ; Ugo: hi, Lucia!
Proto-oo: inheritance

1. Inheritance is not typical of prototype object systems

2. Still, it is used in JavaScript to provide a "more standard" way of reusing code

3. Again, inheritance by delegation:

```
(define (son-of parent)
  (let ((o (new-object)))
    (!! o <<parent>> parent)
    o))
```
basic dispatching:

(define (dispatch object msg)
  (if (eq? object 'unknown)
      (error "Unknown message" msg)
      (let ((slot (hash-ref object msg 'unknown)))
        (if (eq? slot 'unknown)
            (dispatch (hash-ref object '<<parent>>'unknown) msg)
            slot))))
we now have to modify ?? and -> for dispatching

(define-syntx ?? ;; reader
  (syntax-rules ()
    ((_ object msg)
      (dispatch object 'msg)))))

(define-syntx -> ;; send message
  (syntax-rules ()
    ((_ object msg arg ...)
      ((dispatch object 'msg) object arg ...))))
And the example:

(define Glenn (son-of Pino))
(!! Glenn name "Glenn")
(!! Glenn age 50)

(-> Glenn hello Pina) ; Glenn: hi, Lucia!
(-> Glenn ciao)       ; error: Unknown message