

Ch4.1 : Langage Scheme

Prof. Konan Marcellin BROU

marcellin.brou@inphb.ci
2018-2019

Sommaire

- ❑ **Introduction**
- ❑ **Mise en œuvre d'un programme**
- ❑ **Concepts de base**
- ❑ **Les structures de contrôle**
- ❑ **Les fonctions**

I. Introduction

□ 1.1. Présentation

■ 3 catégories de langages de haut niveau :

□ **Programmation Impérative : C/C++, Pascal ...**

- Caractérisée par la présence, dans le langage, d'instructions destinées à modifier l'état du calculateur : les variables, le pointeur d'instructions, la pile, etc.
- Ces langages reposent sur la notion de séquençement.

□ **Programmation Fonctionnelle : Caml, LISP, Scheme ...**

- Style de programmation s'appuyant sur la notion de fonction et d'application de fonction.
- Le but est en fait de proposer un paradigme de programmation, qui soit aussi proche que possible du modèle mathématique.

□ **Programmation Logique : Prolog**

- Tentative d'utiliser la logique comme langage de programmation.
- Une application est programmée sous la forme d'un ensemble de fonctions logiques ou prédicats.

I. Introduction

- **LISP : LIST Processor**

- Créé en 1962 par John Mc Carthy au MIT.
- A la base du développement de l'IA.
- Second plus vieux langage après Fortran.
- MIT : Massachusetts Institute of Technology
- LISP traite des listes.
 - Une liste est une structure de donnée qui commence par une parenthèse ouvrante et qui se termine par une parenthèse fermante.
 - Exemples :

```
()  
(op op1 op2)  
(+ 2 (* 3 5))
```

I. Introduction

- **Les programmes :**

- **Sont conçus à partir de fonctions élémentaires (primitives)**
- **Une combinaison de ces primitives permet d'élaborer des fonctions plus complexes qui seront traitées par l'interprète.**
- **Le Lisp possède environ 500 fonctions primitives dont une vingtaine sont suffisantes pour débiter en Lisp.**

I. Introduction

□ 1.2. Dialectes de LISP

■ Le Lisp ou Le-Lisp

- version française développée à l'INRIA (Institut National pour la recherche en Intelligence Artificielle) par Jérôme Chailloux.

■ Common Lisp

- Le Lisp fut standardisé dans les années 1980-1990
- Common Lisp est le standard international.
- IDE LispStudio
 - <http://fr.foxtoo.com/install-Lisp-studio-10280948.htm>

■ Scheme

- Prononcer Skim
- Créé en 1975 par Steele & Sussmann au MIT.
- Scheme est une version minimale de Lisp.
- <https://download.racket-lang.org/>

■ AutoLISP

- LISP utilisé dans AutoCAD pour la programmation.

I. Introduction

□ 1.4. Scheme

■ Historique

- Skim fut créé en 1975 au MIT (Massachusetts Institute of Technology) par Gerald Jay Sussman et Guy L. Steele.
- Skim est une version minimale de Lisp conçue à des fins pédagogiques.

■ Objectif

- Epurer le LISP de tout en conservant les aspects essentiels, sa flexibilité et sa puissance expressive.
- Ceci fut alors réalisé par une limitation des mots-clefs et une syntaxe extrêmement simple.
- Skim a donc une syntaxe extrêmement simple, avec un nombre très limité de mots-clé.

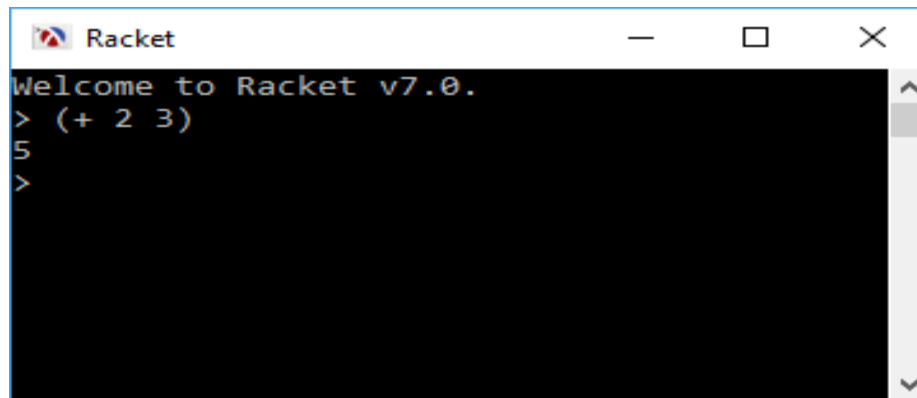
■ Notation préfixée

- Comme en Lisp, la notation préfixée permet de s'affranchir des opérateurs de précedence.

I. Introduction

□ 1.3. Racket

- **Version de Scheme**
- **Langage de programmation de la famille Lisp.**
- **Il fait partie du projet Racket (autrefois PLT Scheme), qui regroupe plusieurs variantes du langage Scheme ainsi qu'une série d'outils pour les utiliser.**
 - <https://download.racket-lang.org/>
- **Utilisation de racket en mode console**
 - **Lancer C:\Program Files\Racket\Racket.exe**



```
Racket
Welcome to Racket v7.0.
> (+ 2 3)
5
>
```


I. Introduction

- **Chargement d'un fichier**

- **Ouvrir une fenêtre de commande**
- **Mettre le chemin de Racket.exe dans le path**

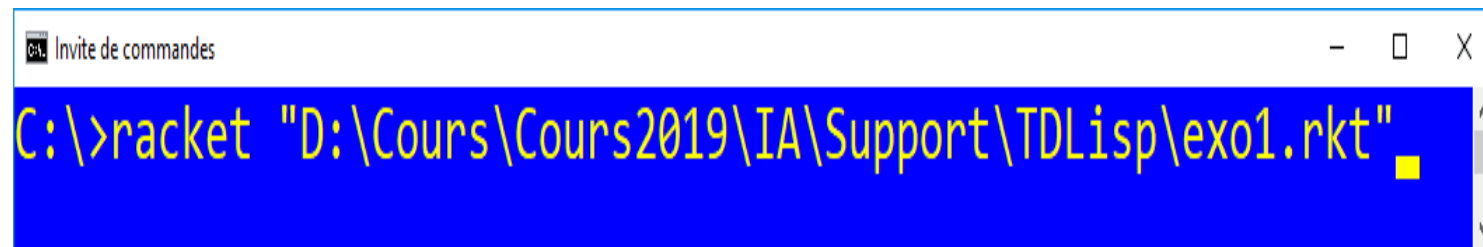


```
C:\>path = %path%;C:\Program Files\Racket.
```

- **Ou Mettre le chemin de Racket.exe dans lea variable d'environnement de Windows**

- **Ouvrir un fichier programme**

- C:\> Racket nomFichier

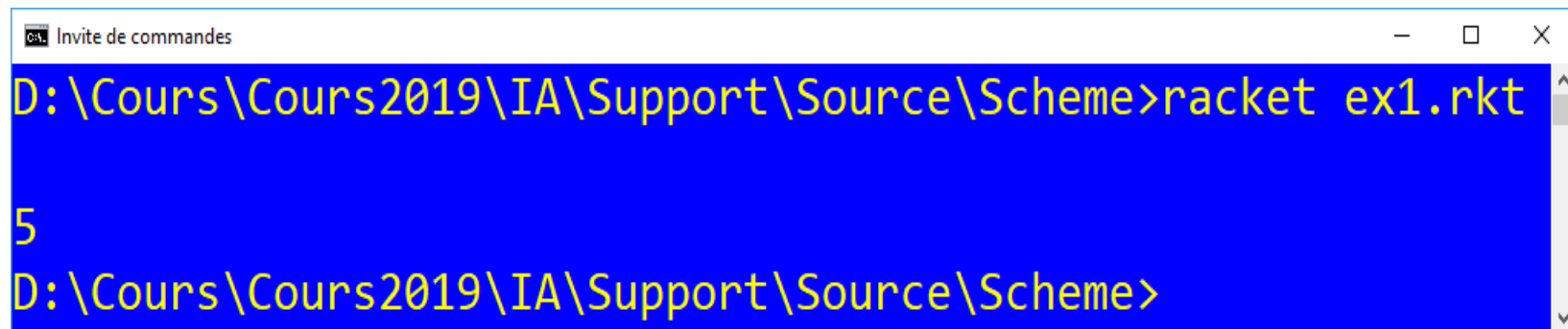


```
C:\>racket "D:\Cours\Cours2019\IA\Support\TDLisp\exo1.rkt"
```

I. Introduction

- ❑ **Le nom du fichier doit être .rkt**
- ❑ **Il faut indiquer la version de Racket utilisée :**
 - #lang racket
 - #lang racket/base
- ❑ **Exemple : ex1.rkt**

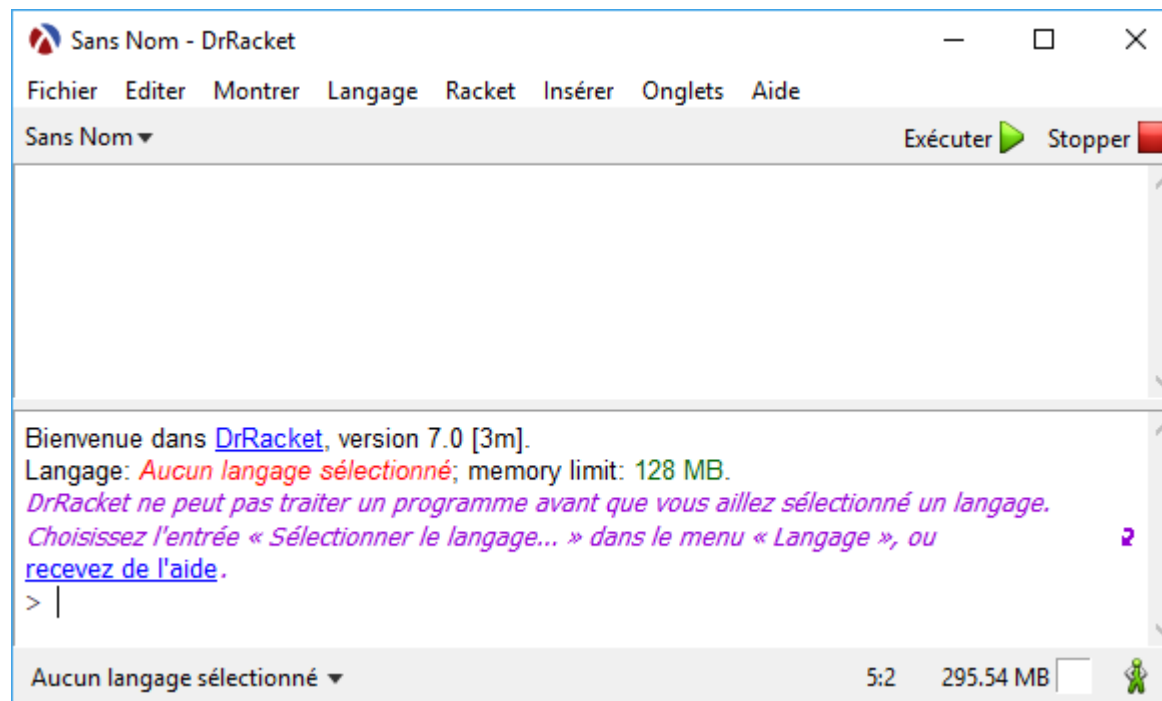
```
#lang racket/base  
(define X 2)  
(define Y 3)  
(print (+ X Y))
```



```
Invite de commandes  
D:\Cours\Cours2019\IA\Support\Source\Scheme>racket ex1.rkt  
5  
D:\Cours\Cours2019\IA\Support\Source\Scheme>
```

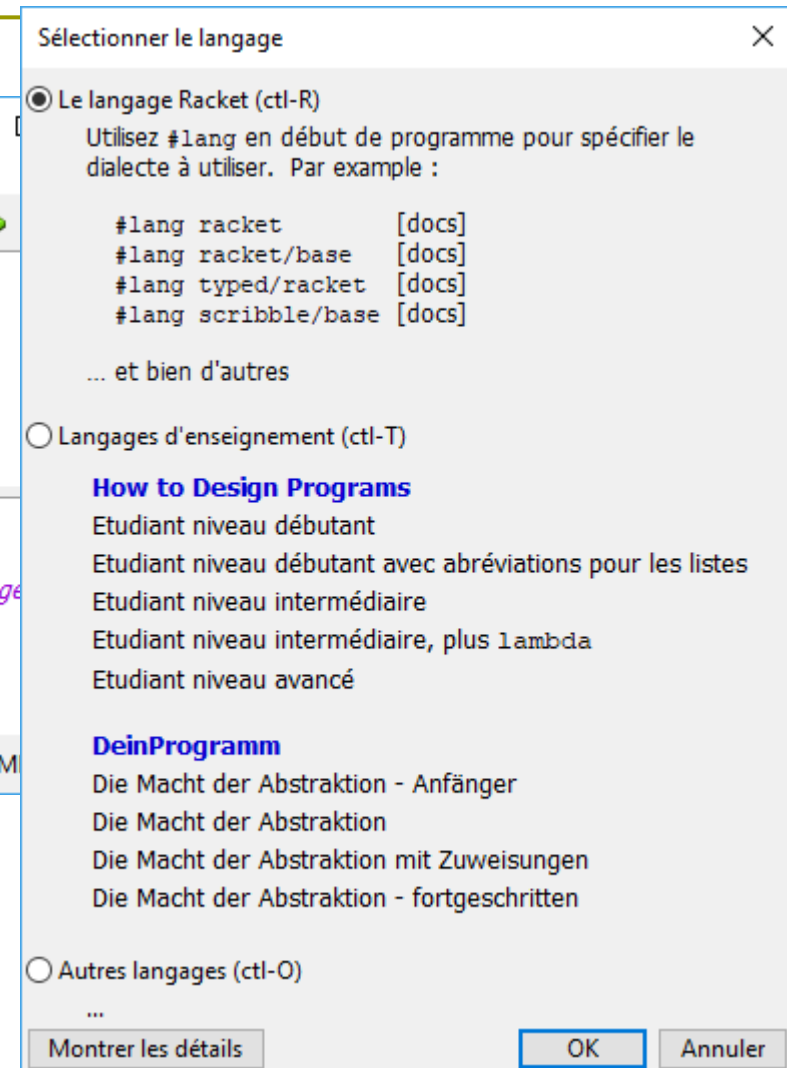
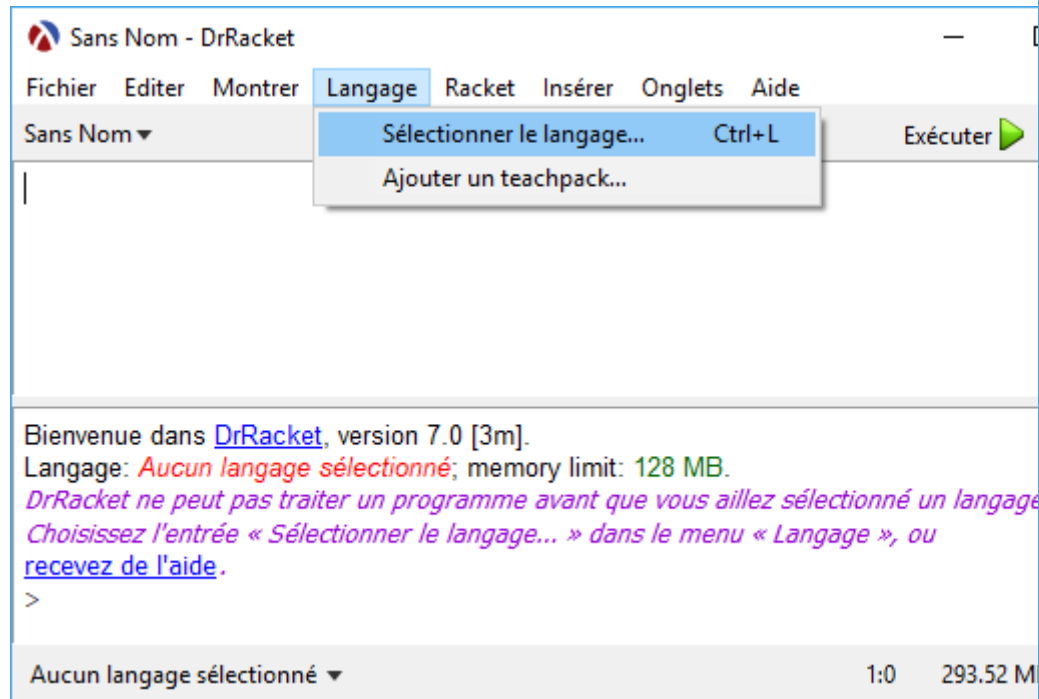
I. Introduction

- **Utilisation de Racket en mode IDE**
 - **Lancer C:\Program Files\Racket\DrRacket.exe**



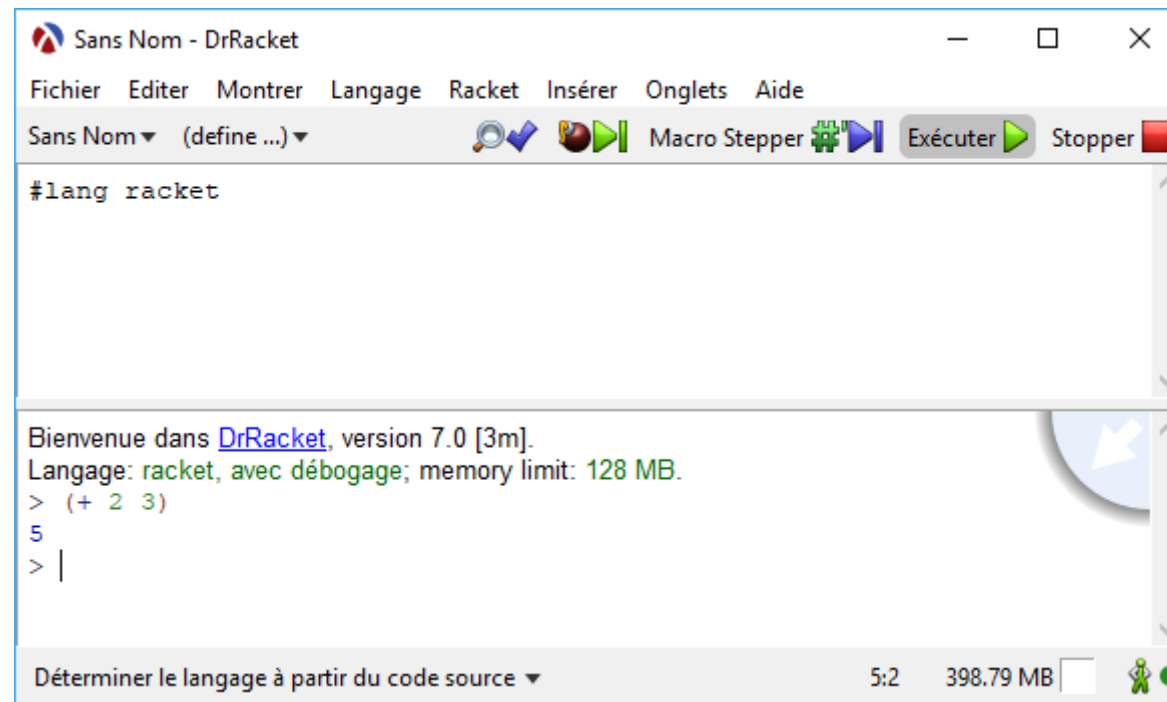
I. Introduction

□ Sélectionner un langage



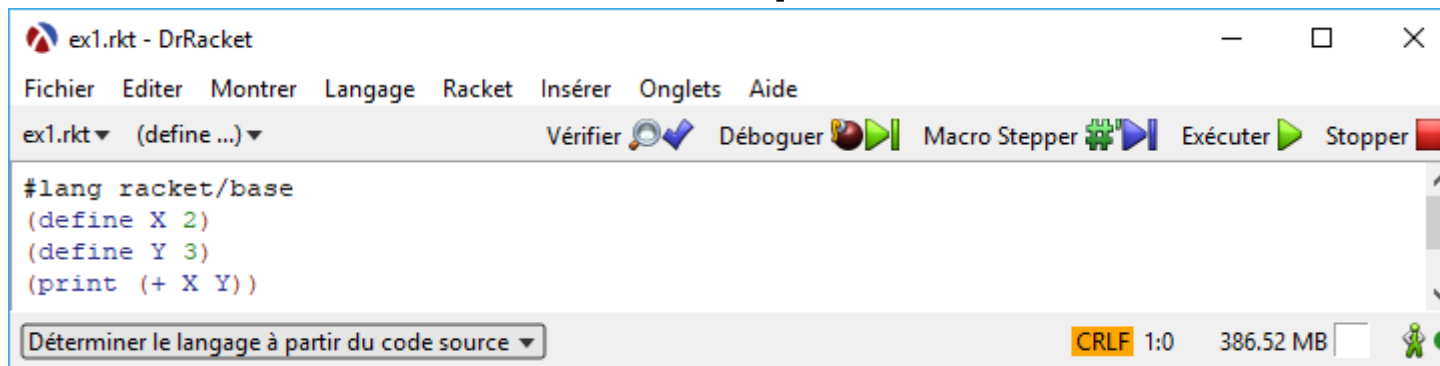
I. Introduction

□ Exécuter un expression

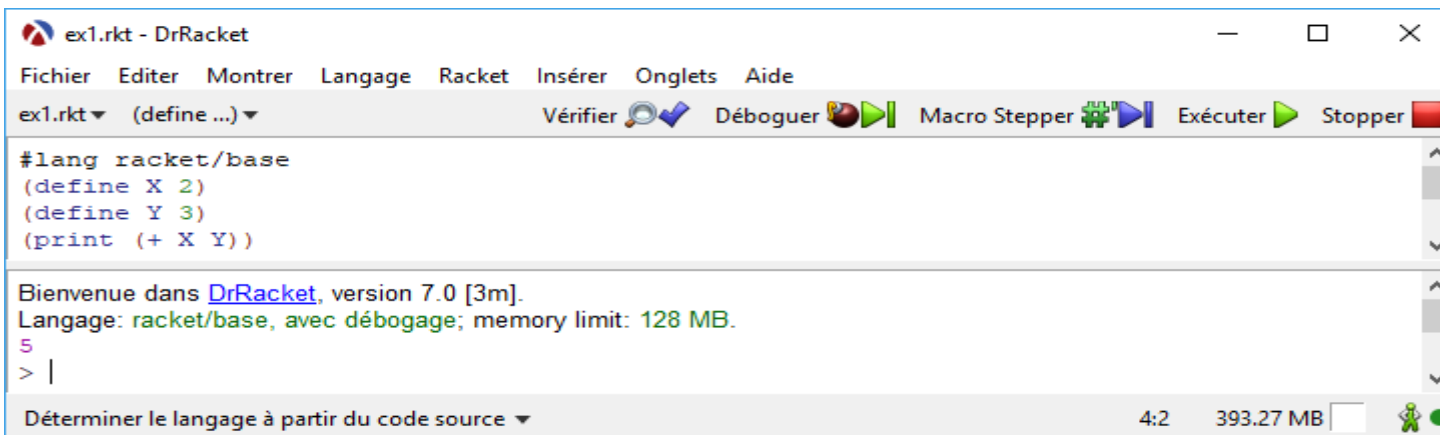


I. Introduction

- **Ouvrir un fichier avec DrRacket**
 - **Menu Fichier/Ouvrir**
 - **Sélectionner le fichier et cliquer sur Ouvrir**

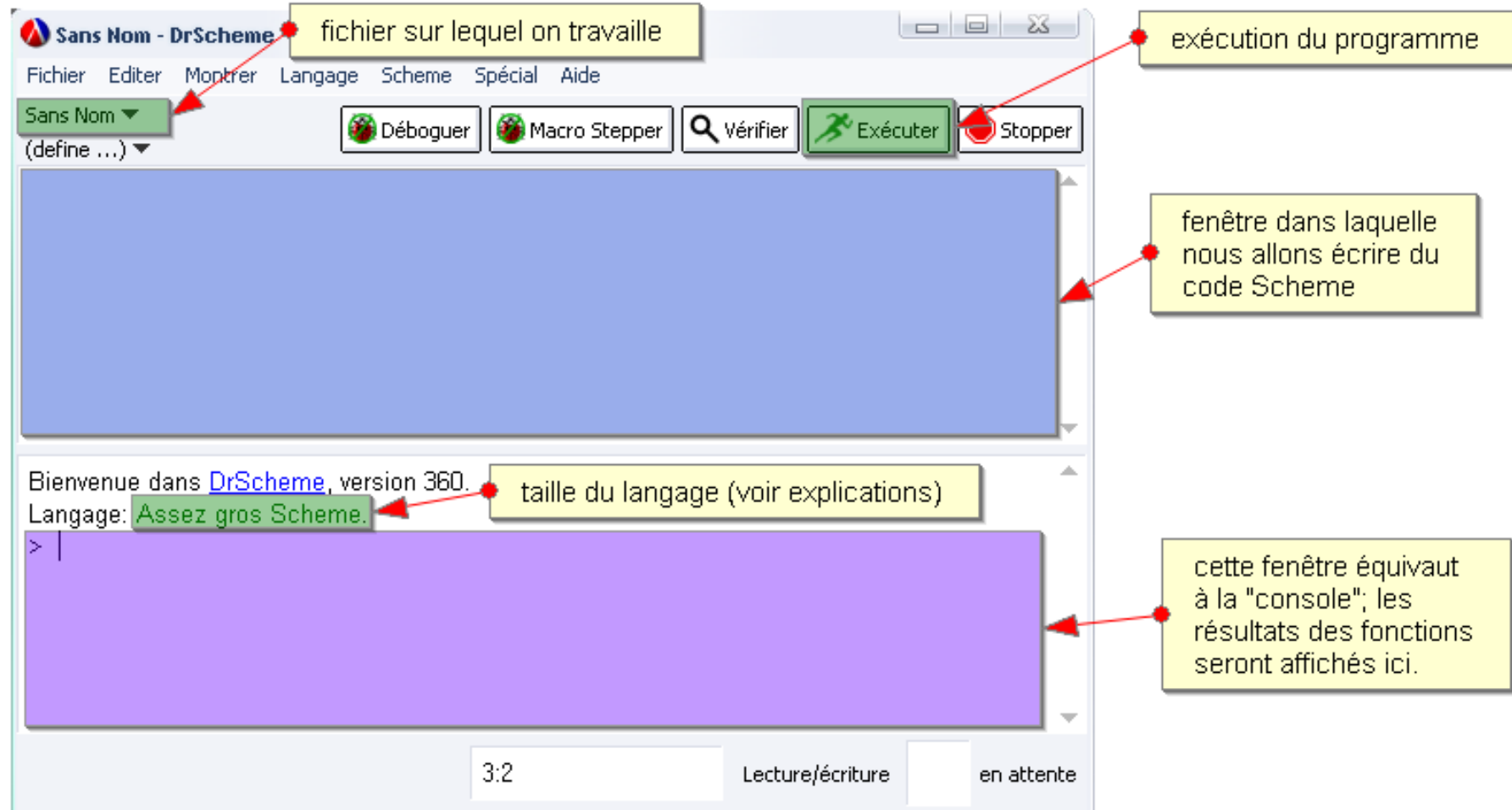


Cliquer sur le bouton Exécuter



I. Introduction

■ Description de l'interface de DrRacket



II. Concepts de base

□ 2.1. Les Expressions Symboliques

■ Liste ::= (élément₁ élément₂ ... élément_n)

- élément₁ : une fonction
- élément₂ ... élément_n : arguments de cette fonction :
 - (fonction argument₁ argument₂ ... argument_{n-1})
- Une telle liste s'appelle une expression symbolique
 - Ou S-Expression
 - Elle représente un appel de fonction.
- Notation préfixée :
 - opérateur avant les opérandes
 - (+ 2 3).

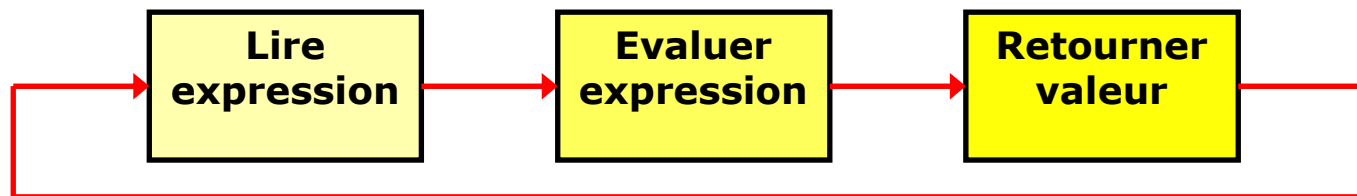
II. Concepts de base

■ Fonctionnement de Scheme

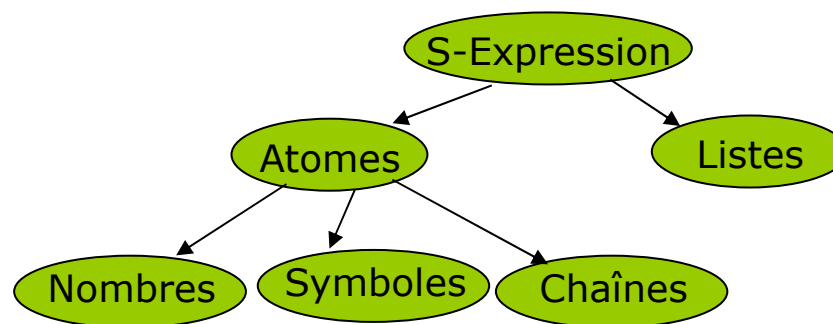
□ Scheme est un interprète :

- qui lit une expression ;
- calcule sa valeur (ça s'appelle évaluer l'expression) ;
- imprime cette valeur ;

□ Représentation :



□ Deux sortes de S-Expression : atome et liste



II. Concepts de base

□ 2.2. Les atomes

- **Suite quelconque de caractères alphanumériques.**

- **Les signes de ponctuation ne peuvent former des atomes.**
 - Certains de ces signes sont utilisés afin d'encadrer des listes.
 - Exemple : la quote (')

- **Exemples :**

- **2009, deuxPi, 18+**

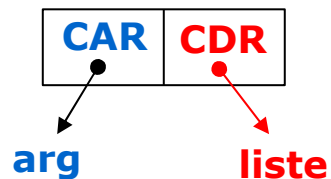
- **Remarque**

- **Scheme respecte la casse**
- **Commentaire : précédé de point-virgule (;)**
 - **;** commence un commentaire à la fin de la ligne.
 - **::** utilisé pour marquer des commentaires plus importants.
 - **#;** commente l'expression s-suivante.

II. Concepts de base

□ 2.3. Les listes

- **Collection d'éléments délimitée par des parenthèses.**
- **Paires en Scheme**
 - **Une paire est une structure d'enregistrement avec deux champs appelés car et cdr**
 - car représente le premier élément
 - cdr représente le second élément
 - **Les champs car et cdr sont accédés par les fonctions car et cdr**
 - **Fabrique en machine, une paire un doublet de pointeurs**



- **Une paire est représentée par la notation**
 - **' (element1 . element2)**
 - le point (.) permet de faire la différence avec une liste de 2 éléments

II. Concepts de base

□ Paire et liste

Liste	Paire				
(a)	<table><tr><td>a</td><td>()</td></tr></table>	a	()		
a	()				
(a . b)	<table><tr><td>a</td><td>b</td></tr></table>	a	b		
a	b				
(a b)	<table><tr><td>a</td><td><table><tr><td>b</td><td>()</td></tr></table></td></tr></table>	a	<table><tr><td>b</td><td>()</td></tr></table>	b	()
a	<table><tr><td>b</td><td>()</td></tr></table>	b	()		
b	()				

- La liste (a b) est en fait une paire de paires.

II. Concepts de base

- **Une liste peut être définie récursivement comme :**
 - soit une liste vide : '()
 - soit une paire dont le cdr est une liste
 - ' (a . '(1 2 3)), qui se réécrit ' (a 1 2 3)
- **Les éléments compris entre parenthèses sont séparés par des espaces.**
- **Exemples :**
 - La liste vide est écrite ()
 - (a b c d e) et (a . (b . (c . (d . (e . ()))))) sont deux notations équivalentes pour une liste de symboles
 - (1 2 3 4)
 - (un 2 trois)
 - (+ 1 (* 2 3))

II. Concepts de base

■ Syntaxe

- **liste** ::= (**{**élément₁ élément₂ ... élément_n**}**)
- **élément** ::= liste | atome
- **atome** ::= nombre | nom
- **nombre** ::= 0 | 1 | 2 | ... | 1024 | ... | -1 | -2 | ... | -1024 | ...
- **nom** ::= suite de caractères sans séparateur
- **séparateur** ::= . | espace | (|) | ' | tabulation | ; | retour chariot

■ Les fonctions de base

- "Faire quelque chose" se dit, en LISP, appeler une fonction.
- Un appel de fonction en LISP se fait de la manière suivante :
 - **(nom-de-fonction {argument1 argument2 ... argumentn})**

II. Concepts de base

□ 2.5. Fonction quote

- Permet de ne pas calculer la valeur d'une expression, mais juste entrer une valeur à utiliser telle quelle.
- L'évaluation d'un appel de cette fonction retourne son argument tel quel.
- Fonction à un seul argument
- Syntaxe
 - **(QUOTE arg1)** Ou **'arg1**
 - Retourne arg1
- Exemples :
 - **(QUOTE Brou)** retourne Brou
 - **(QUOTE (a b c))** retourne (a b c)
 - **'Brou** retourne Brou
 - **'(a b c)** retourne (a b c)

II. Concepts de base

□ 2.6. Fonction Paire

■ (pair? x)

- Retourne #t si x est une paire, #f sinon

■ Exemples :

- (pair? '(a . b)) retourne #t
- (pair? '(a b c)) retourne #t
- (pair? '()) retourne #f
- (pair? "(a b c)") retourne #f

II. Concepts de base

□ 2.7. Fonctions car et cdr

- Permettant d'accéder aux différents éléments d'une liste.
- Ce sont des mnémoniques de l'IBM 704 :
 - **CAR** : **C**ontents of **A**ddress **R**egister
 - **CDR** : **C**ontents of **D**ecrement **R**egister
- **Fonction car**
 - Extrait le premier élément d'une liste.
 - **Syntaxe**
 - **(car uneListe)**
 - le 1^{er} élément de l'argument uneListe doit obligatoirement être une liste.
 - **Exemples**
 - **(car '(a b c))** retourne **a**
 - L'argument (a b c) a été coté car on ne veut pas qu'il soit évalué mais traité comme une liste.
 - **(car '(((o p q r)) s t))** retourne **((o p q r))**

II. Concepts de base

■ Fonction cdr

- **Extrait la liste privée de son premier élément.**

- **Syntaxe**

- **(cdr uneListe)**
- Retourne uneListe privée de son 1^{er} élément
- uneListe doit obligatoirement être une liste.

- **Exemples**

- **(cdr '(a b c))** retourne **(b c)**
- **(cdr '(((o p q r)) s t))** retourne **(s t)**

■ Remarque

- **CAR retourne un atome ou une liste**
- **CDR retourne toujours une liste, même vide.**

■ Combiner CAR et le CDR

- **(car (cdr uneListe)) idem (cadr uneListe)**
- **(car (cdr (cdr uneListe))) idem (caddr uneListe)**

II. Concepts de base

□ **Exemple 1 :**

- **(car (car '(((o p q r)) s t)))** retourne **(o p q r)**
- en effet (car '(((o p q r)) s t)) retourne ((o p q r))
- donc (car '((o p q r))) retourne (o p q r)

□ **Exemple 2 :**

- **(car '(car '(((o p q r)) s t)))** retourne **car**
- en effet '(((o p q r)) s t) retourne ((o p q r)) s t
- donc (car '(car ((o p q r)) s t)) retourne CAR

□ **Exemple 3 :**

- **(cdr (cdr '(a b c)))** retourne **(c)**

□ **Exemple 4 :**

- **(car (cdr '(a b c)))** ou **(cadr '(a b c))** retourne **b**

□ **Exemple 5 :**

- **(car (cdr (cdr '(a b c))))** ou **(caddr '(a b c))** retourne **c**

II. Concepts de base

- **Remarque**

- **cadr** retourne le deuxième élément de sa liste argument.
- **caddr** retourne le troisième élément sa liste argument.
- C.f. Exemple 4 et 5

II. Concepts de base

- **2.7. Fonctions LIST, CONS et APPEND**
 - **Permettent de construire de nouvelles listes.**
 - **Fonction LIST**
 - **Permet de créer une nouvelle liste.**
 - **Syntaxe**
 - **(LIST arg1 arg2 ... argn)**
 - Fabrique une nouvelle liste avec ses argument.
 - **Exemples**
 - **(LIST 1 2 3)** retourne la liste **(1 2 3)**

II. Concepts de base

■ Fonction cons

- Permet de **CON**struire une nouvelle liste en ajoutant un élément au début d'une liste.
- Syntaxe
 - **(cons arg liste)**
 - Retourne une nouvelle liste commençant par la valeur d'argument arg suivie de tous les éléments de la liste liste.
- Exemples
 - **(cons 'a '(b c))** retourne **(a b c)**
 - **(cons '(a b) '(c d))** retourne **((a b) c d)**
 - **(car (cons '((a b c d)) '(e f)))** retourne **((a b c d))**
 - **(cdr (cons '((a b c d)) '(e f)))** retourne **(e f)**
 - **(cons (car '(a b c)) (cdr '(a b c)))** retourne **(a b c)**
- Remarque
 - **(cons (car liste) (cdr liste))** retourne **liste**
 - **(car (cons élément liste))** retourne **élément**
 - **(cdr (cons élément liste))** retourne **liste**

II. Concepts de base

- **Fonction append**

- **Met bout à bout plusieurs listes.**

- **Syntaxe**

- **(append liste1 liste2)**

- Retourne une nouvelle liste commençant par les éléments de la liste liste1 suivie de tous les éléments de la liste liste2.

- **Exemples**

- **(append '(a) '(b c))** retourne **(a b c)**

- **(append '((a b)) '(c d))** retourne **((a b) c d)**

- **(append '() '(a b))** retourne **(() a b))** ou **(NIL a b)**

II. Concepts de base

■ Exercices

Exercice 1 : Déterminez les types des objets ci-dessous

- 45
- (ceci est (un atome))
- (((1) 2) 3) 4)
- -3x
- t
- (((((aiee))))))
-)a
- -45

Exercice 2 : Déterminez le nombre d'éléments des listes suivantes

- (((((aiee))))))
- (1 (2 (3 4)))
- (gee, how easy)
- (une (dernière) exercice)

Exercice 3: Donnez la valeur des appels de la fonction quote ci-dessous

- (quote tiens)
- 'tiens
- '(1 (2 (3 4)))
- "tien

Exercice 4 : Quelles valeurs retournent les expressions suivantes

- (car (cdr (car '((un one) (deux two) (three trois))))))
- (car (cdr (cdr (cdr '(1 2 3 4 5 6)))))
- (cdr '(car (cdr '(1 2 3 4))))
- (cdr '(((1) 2) 3))

Exercice 5 :Donnez les expressions nécessaires pour retourner l'atome **a** des listes suivantes

- (b a c)
- (((a) b) c) d)
- (d (c (b (a))))
- (1 (2 (3 a) 4) 5)

Exercice 6 : Quelles valeurs sont retournées par les expressions suivantes ?

- (cons 'nobody (cons 'is '(perfect)))
- (cons (cons (cons 'a nil) (cons 'b nil)) (cons 'c nil))
- (cadr (cons 'a (cons 'b (cons 'c nil))))
- (cons nil (cons nil (cons nil nil)))

II. Concepts de base

Exercice 7 : Donnez les expressions qui, à partir de l'atome **a**, permettent de construire les listes suivantes. Exemple : la liste **(a)** peut être construite par l'expression **(cons 'a nil)**.

((a))

(a a)

(a (a) a)

(nil a)

II. Concepts de base

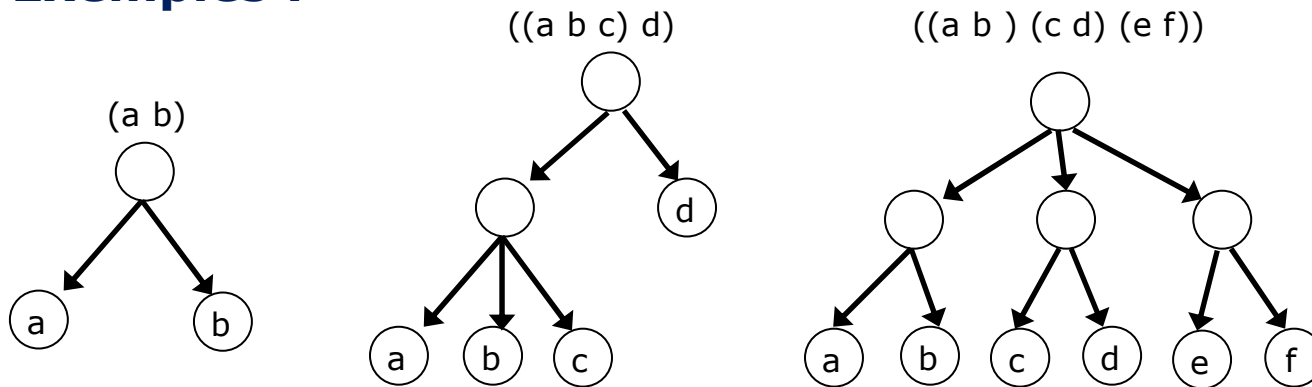
□ 2.8. Arbre de représentation des listes

■ On peut visualiser une listes sous forme d'un arbre.

■ Principe

- Descendre d'un niveau à la rencontre d'une parenthèse ouvrante.
- Remonter d'un niveau à la rencontre une parenthèse fermante.

■ Exemples :



■ Remarque

- Cercle vide : représente une liste ou sous-liste.
- Cercle étiqueté : éléments d'une liste directement connectés au cercle de la liste.

II. Concepts de base

■ Exercice 1 :

Dessinez les arbres correspondants aux listes ci-dessous.

((a))

(a a)

(a (a) a)

(nil a)

II. Concepts de base

- **Exercice 2 :**

- **Donnez l'arbre de l'expression suivante :**
 - **$((a)(b)(c))\ d\ (((e\ f))\ g)\ h)))$**

II. Concepts de base

□ 2.9. Fonctions de test

■ (list? x)

- Retourne #t si x est une liste, #f sinon

- Exemples :

- (list? '(a b c)) donne #t
- (list? '()) donne #t
- (list? '(a . b)) donne #f

■ (null? x)

- Retourne #t si x est la liste vide, #f sinon

■ (eq? exp1 exp2) idem (equal? exp1 exp2)

- Cette opération teste l'identité de ses arguments.

- Retourne #t si x est une liste, #f sinon

- Exemples :

- (eq? '(a b c) '(a b c)) donne #t
- (eq? 2 2) donne #t
- (eq? "Hello" "Hello") donne #t

II. Concepts de base

□ 2.10. Les variable

- **Un identificateur sert à designer une valeur, à nommer une fonction.**
- **Une variable possède un identificateur qui désigne une zone de mémoire contenant une information d'un type donné.**
 - **Sa valeur pouvant être modifiée au cours de l'exécution du programme.**
 - **Respect de la casse.**
- **Il existe des variables prédéfinies**
 - **Exemple :**
 - `> pi`
 - `3.141592653589793`
- **Remarque**
 - **Si on redéfinit un nom déjà défini, alors c'est la dernière définition qui compte.**

II. Concepts de base

- **2.11. Déclaration de variable**
 - **Scheme utilise un typage dynamique**
 - **Fonction define**
 - **Permet de déclarer une variable et de l'initialiser**
 - **Prend deux arguments.**
 - Le premier doit être le nom d'une variable
 - Le second une expression LISP quelconque.
 - **Syntaxe**
 - **(define variable valeur)**
 - **variable est le résultat de l'évaluation de valeur**
 - **Exemples**
 - **(define X 2) ; déclare X et l'initialise à 2**
 - **(define Y (* 2 3)) ; déclare Y et l'initialise à 2*3=6**

II. Concepts de base

□ 2.12. Opérateurs

■ Opérateurs arithmétiques.

□ **+, -, /, *, modulo, quotient, remainder (idem modulo)**

□ **Exemples :**

- (+ 2 3) donne 5
- (+ 1 2 3 4) donne 10
- (/ 5 2) donne 2.5
- (modulo 5 2) donne 1
- (**remainder** 5 2) donne 1
- (quotient 5 2) donne 2

■ Opérateurs relationnels et logiques :

□ **Ces opérateurs retournent #T pour vrai et #F pour faux.**

□ **>, >=, <, <=** tests de supériorité et d'infériorité

□ **=, <>** tests d'égalité ou d'inégalité

□ **AND, OR** ET et OU logiques

□ **NOT** Non

II. Concepts de base

□ 2.13. Les types

■ Les types de données de base :

□ Les booléens

- Deux littéraux booléens: #t pour true et #f pour false
- (booléen? x) indique si x est une valeur booléenne.

□ Les nombres, qui peuvent être :

- Entiers de taille indéfinie : (+ 2 3)
- Réels : (- 5.3 2.4), -inf.0, +inf.0, nan.o
- Complexes : 2+3i
- Fonctions de test : number? integer? complex? rational? real? byte?
- (integer? 2)

II. Concepts de base

■ Opération sur les nombre

expt	élévation à la puissance exemple : (expt a b) calcule a^b
sqrt	racine carrée, exemple (sqrt x).
abs	valeur absolue, exemple (abs x).
sin, cos, tan	fonctions trigonométriques, exemple (sin x).
asin, acos, atan	leurs fonctions réciproques.
log	logarithme néperien, exemple (log x).
exp	Exponentielle, exemple (exp x) calcule e^x .

II. Concepts de base

■ Nombres complexes

- **(make-rectangular Prel Pimg) :**
 - permet de créer un nombre complexe $Pr + iPimg$
- **(real-part Z) : Z contient la partie réelle**
- **(imag-part Z) : Z contient la partie imaginaire**
- **Exemple :**

```
(define Z (make-rectangular 0 0))  
(set! Z (make-rectangular 2 3))  
(write Z) ; affiche 2 +3i  
(write (real-part Z)) ; affiche 2  
(write (imag-part Z)) ; affiche 3
```

□ Opérations arithmétiques

- On peut utiliser les opérateurs : $+$, $-$, $*$, $/$
- Exemple :

```
(set! Z1 (make-rectangular 2 3))  
(set! Z2 (make-rectangular 12 13))  
(set! Z (+ Z1 Z2))  
(write Z) ; affiche 14 +16i
```

II. Concepts de base

■ Les caractères

□ Les caractères sont des valeurs scalaires Unicode

□ Exemples :

- `#\a` ; un "a"
- `#\A` ; un "A"
- `#\.` ; un point (est aussi un caractère)
- `#\space` ; un espace
- `#\newline` ; aller à la ligne

□ Opérateurs qui respectent la casse :

- `char>?`, `char=>?`, `char<?`, `char<=?` et `char=?`
- utilisés pour comparer deux caractères par ordre alphabétique.
- Retourne un booléen (`#t` ou `#f`).

□ Exemple :

- `(char<? #\f #\g)`
- renverra `#t`, `f` est plus petit que `g` dans l'ordre alphabétique.

□ Opérateurs qui ne tiennent pas compte de la casse

- `char-ci=?`, `char-ci<?`, `char-ci>?`, `char-ci<=?`, `char-ci>=?` .

II. Concepts de base

■ Les chaînes de caractères

□ Les chaînes de caractères sont entre guillemets

- "Bonjour Fatou"
- (string-append "Bonjour" " ", " "Fatou") donne "Bonjour Fatou"
- (string-length "Bonjour Fatou") donne 13

□ Remarque

- "a" n'est pas la même chose que #\a, une chaîne peut contenir un seul caractère.
- Pour utiliser un " dans une chaîne on placera un \ devant :
- "Voici une \"chaîne\"".
- Et pour utiliser un \ il suffira de placer un autre \ devant :
- "Voici un \\ dans une chaîne".

□ Comparaison de chaînes :

- On utilisera string<?, string>?, string=>?, string=<?, string=?
- Retourne un booléen selon la longueur des deux chaînes considérées.

II. Concepts de base

- **Concaténation de chaînes :**

- Permet de mettre deux chaînes bout à bout grâce à l'opérateur : string-append.
- (string-append "Toto " " Ali") donne "Toto Ali«

- **Conversion**

- **Formes générales**

- integer->char
- char->integer
- number->string
- string->number
- real->decimal-string

II. Concepts de base

□ Exemple

- (char->integer #\a) donne 97
- (integer->char 97) donne #\a
- (string->number "97") donne 97
- (number->string 97) donne "97"
- (real->decimal-string pi) donne "3.14"
- (real->decimal-string pi 5) donne "3.14159"

II. Concepts de base

- **Les Vecteurs**

- **Les vecteurs en Scheme sont représentés par une liste d'éléments, précédée par le caractère spécial #.**
- **Exemples :**
 - **`#(3 4 5)`**
 - **`#("text1" 3 #\a)`**

II. Concepts de base

■ Les listes d'associations

- Une liste d'association est une liste de couples clé/valeur

- Dictionnaire.

- Exemple :

```
(define client
  '( (1 ("Toto" "Yakro"))
      (2 ("Froto" "Abidjan"))
      (3 ("mankou" "Daloa"))
      (4 ("Fatou" "Man"))
    )
)
```

- Accès au éléments

- Fonction assoc : retourne #f si rien n'est associé à la clé, sinon retourne le couple clé/valeur.
 - Exemple

```
(assoc '2 client) donne '(2 ("Froto" "Abidjan"))
```

II. Concepts de base

- **Le résultat retourné par assoc est une liste**
 - Les opérateurs sur les listes peuvent être utilisés.
 - Exemples :

```
(car (assoc '2 client)) donne la clé 2
(cdr (assoc '2 client)) donne la valeur '("Froto" "Abidjan")
(caadr (assoc '2 client)) donne le nom "Froto«
(car (cdr (car (cdr (assoc '2 client))))) donne "Abidjan"
idem
(cadr (cadr (assoc '2 client))) donne "Abidjan"
```

II. Concepts de base

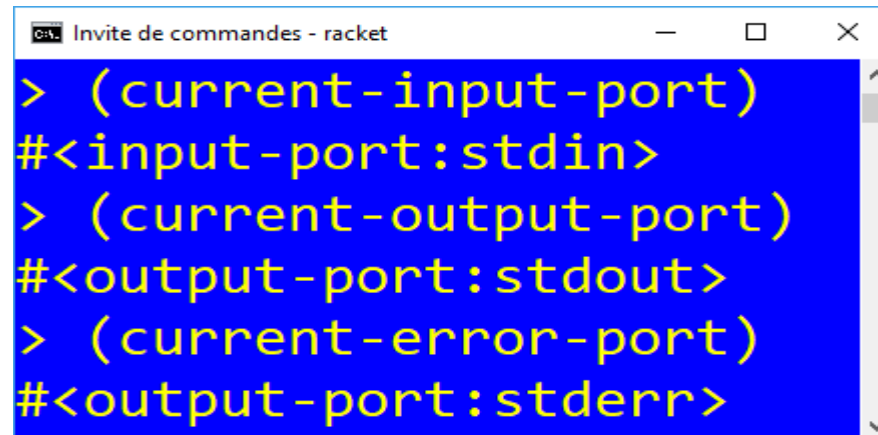
□ 2.14. Les entrées/sorties

- **Ces opérations s'effectuent sur des ports, qui peuvent être**
 - liés au terminal (prédéfinis stdin, stdout ou stderr) ;
 - mais aussi à des fichiers ou des objets du langage, les chaînes de caractères.
- **Les ports**
 - Ce sont des objets du langage sur lesquels des opérations de lecture/écriture peuvent être effectuées.

II. Concepts de base

□ Ports prédéfinis

- (current-input-port) : port d'entrée associé à stdin
- (current-output-port) port de sortie associé à stdout
- (current-error-port) port de sortie d'erreurs associé à stderr



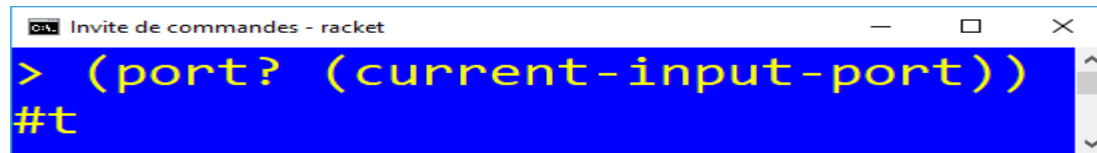
A screenshot of a Racket command prompt window titled "Invite de commandes - racket". The window has a blue background and yellow text. It shows three commands and their corresponding outputs:

```
> (current-input-port)
#<input-port:stdin>
> (current-output-port)
#<output-port:stdout>
> (current-error-port)
#<output-port:stderr>
```

II. Concepts de base

□ (port? objet)

- Retourne un booléen, indiquant si le paramètre est un port d'E/s.



```
Invite de commandes - racket
> (port? (current-input-port))
#t
```

□ (input-port? objet)

- Retourne un booléen, indiquant si le paramètre est un port d'E/S susceptible de délivrer des caractères.



```
Invite de commandes - racket
> (input-port? (current-input-port))
#t
```

□ (output-port? objet)

- Retourne un booléen, indiquant si le paramètre est un port d'E/S susceptible de recevoir des caractères.



```
Invite de commandes - racket
> (output-port? (current-output-port))
#t
```

II. Concepts de base

- **Lecture : read**

- Permet de lire une expression au clavier
- Ne prend pas d'argument
- Syntaxe
 - **(read [port])**
- Exemple
 - **(read)**
 - **(+ (read) (read))**
 - 2** on saisit 2 et on valide
 - 3** on saisit 3 et on valide
 - 5** résultat

II. Concepts de base

■ **Affichage :**

□ **display**

- Permet d'afficher une expression à l'écran pour que un **humain** puisse la lire.
- Syntaxe
 - **(display arg)**
- Exemples :
 - **(display 2) donne 2**
 - **(display "Bonjour Fatou"))** donne **Bonjour Fatou**

□ **write**

- Permet d'afficher une expression à l'écran lisible par **(read)**.
- Son argument est affiché telquel.
- Syntaxe
 - **(write arg)**
- Exemple
 - **(write 2) donne 2**
 - **(write "Bonjour Fatou")** donne **"Bonjour Fatou"**

II. Concepts de base

□ Différence entre display et write

- display est similaire à write, mais affiche directement les chaînes et les caractères.
- Les chaînes sont imprimées sans guillemets ni barres obliques et les caractères sont imprimés sans #\.
- Exemples :
 - **(display "(a b c)") et (display '("a b" c))** donne **(a b c)**.
 - **(write "(a b c)")** donne **"(a b c)"**
 - **(write '("a b" c))** donne **("a b" c)**.
- De ce fait, display ne doit pas être utilisé pour imprimer des données en attente de lecture avec read.
- Il plutôt faut utiliser write

II. Concepts de base

- **newline**

- Permet de sauter une ligne
- Exemple
 - **(print "Bonjour Fatou"))**
 - **(newline)**

- **Print (pas normalisé)**

- Permet d'afficher une expression à l'écran
- Syntaxe
 - **(print arg)**
- Exemple
 - **(print 2) donne 2**
 - **(print "Bonjour Fatou") donne "Bonjour Fatou"**

II. Concepts de base

□ **Printf (pas normalisé)**

- Permet d'afficher une expression à l'écran
- Syntaxe
 - **(printf <format-string> v1 v2 ...)**
 - format-string : composé de ~a (display) et ~s (write), et sauts \n
- Exemple
 - **(define X 2)**
 - **(define Y 3)**
 - **(printf "x=~a Y=~a" X Y)** donne x=2 Y=3
 - **(printf "x=~s Y=~s" X Y)** donne x=2 Y=3

II. Concepts de base

- **Concaténation**

- **(define x 2)**
- **(print (string-append "x = " (number->string x)))**
- Donne "x = 2"

III. Les structures de contrôle

□ 3.1. Affectation

- **Attribuer une nouvelle valeur à un symbole.**

- **Fonction set!**

- **Affecte une valeur à une variable**

- **Prend deux arguments.**

- Le premier doit être le nom d'une variable

- Le second une expression LISP quelconque.

- **Syntaxe**

- **(set! variable valeur)**

- variable est le résultat de l'évaluation de valeur

- **Exemples**

- **(set! X 2)** affecte 2 à X

- **(set! X (* 2 3))** affecte $2*3=6$ à X

III. Les structures de contrôle

- **Exercice :**
 - **échanger les valeurs numérique de X=2 et Y=3 sans passer par une variable intermédiaire.**

III. Les structures de contrôle

▣ 3.2. L'alternative

- Suivant les résultats de l'application des tests, sélectionner entre différentes expression celle qui est adéquate.
- Remarque
 - Vrai = #t (abréviation de True)
 - Faux = #f (abréviation de False)
- Fonction IF
 - Syntaxe

```
(if condition action1-si-vrai  
                action2-si-faux  
            )
```
 - Exemple
 - (define X 2)
 - (if (> X 0) (print "Positif") (print "Négatif"))

III. Les structures de contrôle

- **Remarque**

- Exécuter plusieurs expressions dans action
- On utilise la fonction **begin**
- Exemple
 - (define X 2)
 - (if (> X 0) (**begin (print X) (print "Positif")**) (print "Négatif"))

III. Les structures de contrôle

■ Fonction cond

- Permet de faire un choix multiple.
- Généralisation du IF (IF imbriqués)
- Syntaxe

```
(cond (condition1 action1-si-vrai)  
      (condition2 action2-si-vrai)  
      ...  
      (conditionn actionn-si-vrai)  
      (else actionm)  
)
```

□ Exemple

```
(define X 2)  
(cond ((> X 0) (print "Positif"))  
      ((< X 0) (print "Négatif"))  
      (else (print "Zéro"))  
)
```


III. Les structures de contrôle

■ Exercices

- **Ecrire une S-Expression permettant de déterminer le maximum de 3 nombres réel saisis au clavier.**
- **Ecrire de deux façons différentes (if et cond) une S-Expression permettant de simuler une calculatrice effectuant les opérations arithmétiques élémentaires (+, -, *, /). Les opérandes et l'opérateur sont entrés au clavier.**

III. Les structures de contrôle

□ 3.3. Les boucles

- Les structures de contrôle **while**, **repeat-until** et **for** qui existent dans la plupart des langages de programmation, sont absentes de Scheme.
- Elles sont remplacées par des appels de fonction récursifs.
- Il est toutefois possible de définir ces constructions à l'aide de macros.

III. Les structures de contrôle

■ Boucle for

□ Syntaxe :

- **(for ([i N]) (corps de la boucle))**
- I est la variable de la boucle.
- N est le nombre d'itération.

□ Exemple 1 : afficher les 4 premiers nombres

- Valeur initiale de i = 0

#lang racket/base	i=0
(for ([i 4])	i=1
(printf "i=~a\n" i)	i=2
)	i=3

□ Exemple 2 : afficher les 4 nombres

- Définir la valeur initiale de i

#lang racket/base	i=2
(for ([i (in-range 2 6)])	i=3
(printf "i=~a\n" i)	i=4
)	i=5

III. Les structures de contrôle

□ Exemple 3 : pas décroissant

#lang racket/base	3
(for ([i (in-range 3 -1 -1)])	2
(printf "i=~a\n" i)	1
)	0

- 1^{er} -1 : valeur finale, 2^{ème} -1 : pas

□ Exemple 4 : Parcourir une liste

#lang racket/base	a
(for ([i (in-list '(a b c d))])	b
(displayln i)	c
)	d

□ Exemple 5 : Parcourir un vecteur

#lang racket/base	a
(for ([i (in-vector #(a b c d))])	b
(displayln i)	c
)	d

III. Les structures de contrôle

■ Boucle for*

□ Syntaxe :

- **for* ((variable init [step])...) (test expr...) body...**
- Lier les variables et évaluer le corps jusqu'à ce que le test soit vrai.
- La valeur de retour est la dernière expression après test, si elle est donnée.

□ Exemple 1 : afficher "Bonjour Fatou" 5 fois

<pre>#lang racket/base (define (affiche n) (for* ([i (in-range n -1 -1)]) (print "Bonjour Fatou") (newline))) (affiche 5)</pre>	<pre>"Bonjour Fatou" "Bonjour Fatou" "Bonjour Fatou" "Bonjour Fatou" "Bonjour Fatou"</pre>
---	--

III. Les structures de contrôle

■ Boucle do

□ Syntaxe :

- **do ((variable init [step])...) (test expr...) body...**
- Lier les variables et évaluer le corps jusqu'à ce que le test soit vrai.
- La valeur de retour est la dernière expression après test, si elle est donnée.

□ Exemple 1 : afficher le 4 premiers nombres

<pre>(do ((i 1 (+ i 1))) ((> i 4)) (display i))</pre>	1234
---	------

III. Les structures de contrôle

- **Exemple 1 : afficher les éléments d'une liste (boucle1.lisp)**
 - Si la liste est vide on n'affiche rien
 - Sinon on affiche le car de la liste et on refait la même chose avec le cdr de la liste.
 - Fonction atom teste si son argument un atom

<pre>(define atom? (lambda (v) (not (pair? v))))</pre>	<pre>> (atom? 1) #t > (atom? '()) #t > (atom? '(1 2 3)) #f ></pre>
--	--

III. Les structures de contrôle

- Fonction affiche

<pre>(define affiche (lambda (liste) (cond ((atom? liste) (display "")) (else (display (car liste)) (display " ") (affiche (cdr liste))))))</pre>	<pre>> (affiche '(1 2 3)) 1 2 3</pre>
---	--

III. Les structures de contrôle

■ Opérateurs map et for-each

□ for-each

- évalue la fonction donnée sur les éléments de liste de gauche à droite et ignore la valeur de retour de la fonction.
- C'est idéal pour faire des choses secondaires sur chaque élément de la liste.

□ map

- Applique une fonction successivement à une liste d'arguments.
- évalue la fonction donnée sur les éléments de la liste sans ordre spécifique et enregistre la valeur de retour de la fonction pour la renvoyer à l'appelant.
- C'est idéal pour effectuer un traitement purement fonctionnel sur chaque élément de la liste.

□ Remarque

- Si la valeur de retour de map ne va pas être utilisée, il est préférable d'utiliser for-each à la place.
- De cette façon, il n'est pas nécessaire de collecter les valeurs de retour des appels de fonction.

III. Les structures de contrôle

□ Opérateur for-each

- for-each simple : affichage des éléments d'une liste

<pre>(for-each (lambda (x) (display (* x 1)) (display " "))) '(1 2 3))</pre>	<pre>1 2 3</pre>
--	------------------

- for-each dans une fonction : liste à afficher passé en paramètre

<pre>(define affiche2 (lambda (liste) (for-each (lambda (x) (display (* x 1)) (display " "))) liste))</pre>	<pre>> (affiche2 '(1 2 3)) 1 2 3</pre>
--	---

III. Les structures de contrôle

□ Opérateur map

(map sqrt '(1 2 3 4))	'(1 1.4142135623730951 1.7320508075688772 2)
-----------------------	---

- map simple : affichage des éléments d'une liste

(map (lambda (x) (* x 1)) '(1 2 3))	'(1 2 3)
---	----------

- map dans une fonction : liste à afficher passé en paramètre

(define affiche3 (lambda (liste) (map (lambda (x) (* x 1)) liste)))	> (affiche3 '(1 2 3)) '(1 2 3)
--	-----------------------------------

III. Les structures de contrôle

□ Exemple 2 : Afficher "Bonjour Fatou" n fois (boucle2.lisp)

<pre>(define afficheF (lambda (n) (cond ((= n 0) (display "")) (else (display "Bonjour Fatou") (newline) (afficheF (- n 1)))))</pre>	<pre>> (afficheF 3) Bonjour Fatou Bonjour Fatou Bonjour Fatou ></pre>
--	---

III. Les structures de contrôle

■ Exercices

- **Exercice 1 : Ecrire une S-Expression permettant de calculer le factoriel d'un nombre lu au clavier.**
- **Exercice 2 : Ecrire une S-Expression pour exécuter l'exercice 1 autant de fois que l'utilisateur le désire.**

III. Les structures de contrôle

- **Correction Exercices**

- **Exercice 1 : exoFact1.rkt**
- **Exercice 2 : exoDoFac.rkt**

IV. Les Fonctions

□ 4.1. Fonction nommée

- La définition d'une fonction se fait a l'aide du mots clefs : **define.**
- **Syntaxe**

```
(define (f x1 ..... xn)  
  e1 ..... em  
)
```

f : nom de la fonction
x₁ x_n : paramètres
e₁ e_m : corps de la fonction

- **Appel de la fonction**
 - **(nomFonction param1 param2 ...)**

IV. Les Fonctions

■ Exemple 1 : somme de 2 nombres

```
(define (somme n m)
  (+ n m)
)

> (somme 2 4)
6
```


IV. Les Fonctions

□ 4.1. Fonction anonyme

- **La définition d'une fonction se fait à l'aide du mot clef : lambda.**
- **Lambda Signifie que l'on définit une fonction anonyme.**
 - **Des objets qui s'évaluent à une fonction, sans nécessairement être liés à une quelconque variable.**
- **Syntaxe**

```
(lambda (param1 param2 ...)  
  (le corps de la fonction)  
)
```

- **Appel de la fonction**
 - **(nomFonction param1 param2 ...)**

IV. Les Fonctions

■ Exemple 1 :

(lambda () "Bonjour Fatou")

Donne #<procedure>

idem

(λ () "Bonjour Fatou")

□ Appel

((lambda () "Bonjour Fatou"))

Donne **"Bonjour Fatou"**

IV. Les Fonctions

■ Exemple 2

```
(lambda (x y)
  (+ x y)
)
```

Donne #<procedure>

```
((lambda (x y)
  (+ x y)
)
 2 3
)
```

Donne 5

IV. Les Fonctions

- **Lambda calcul**

- ou λ -calcul
- **Système formel à la base du concept de fonction.**
- **On y manipule des expressions appelées λ -expressions**
- **La lettre grecque λ est utilisée pour lier une variable.**
 - Exemple : si **M** est une λ -expression, **$\lambda x.M$** est aussi une λ -expression et représente la fonction qui à x associe M.
- **Premier formalisme pour définir et caractériser les fonctions récursives : il a donc une grande importance dans la théorie de la calculabilité.**
- **Source : <https://fr.wikipedia.org/wiki/Lambda-calcul>**

IV. Les Fonctions

□ 4.1. Fonction anonyme nommée

- La définition d'une fonction se fait a l'aide de deux mots clés : **lambda** et **define**.
- **Lambda** Signifie que l'on définit une fonction sans nom ou anonyme.
- **Syntaxe**

```
(define nomFonction (lambda (param1 param2 ...)  
                      (le corps de la fonction)  
                      )  
)
```

- **Appel de la fonction**
 - **(nomFonction param1 param2 ...)**

IV. Les Fonctions

- **Exemple**

- **Soit la fonction fois2 : $X \rightarrow 2*X$**

```
(define fois2 (lambda(x)
  (* 2 x)
  )
)
```

Idem

```
(define (fois2 x)
  (* 2 x)
)
```

- **Appel de la fonction**
 - **(fois2 3)**

IV. Les Fonctions

□ 4.2. Fonctions sans argument

■ Syntaxe

```
(define nomFonction (lambda ()  
                      (le corps de la fonction)  
                      )  
)
```

■ Exemple

□ Soit la fonction **deuxpi** : $\rightarrow 2 \times 3.14$

```
(define deuxpi (lambda()  
  (* 2 pi)  
  )  
)
```

■ Appel de la fonction

□ **(deuxpi)**

IV. Les Fonctions

■ Exercices

- **Exercice 1 : Ecrire une fonction qui calcule la somme des éléments d'une liste.**
- **Exercice 2 : Ecrire une fonction qui fait la somme des n premiers nombres pairs.**
- **Exercice 3 : Ecrire une fonction qui calcule la somme des éléments de deux listes.**
- **NB : version itérative et version récursive**

IV. Les Fonctions

□ 4.3. Variable globale et variable locale

■ Variable globale

- Les variables (dont les fonctions) définies avec **define** sont globales, c'est-à-dire qu'elles existent en dehors de l'expression qui les a créées.

■ Variable locale

- Si une variable ne doit être utilisée que dans une expression et pas ailleurs, il faut la définir comme variable locale :
 - La mémoire allouée est libérée une fois l'expression évaluée ;
 - on peut réutiliser le nom de la variable ailleurs sans risque de collision (cas de noms classiques comme « i », « x », ...) ;
 - Cela facilite la récursivité : un même nom de variable est utilisé à « plusieurs niveaux » et peut avoir une valeur différente à chaque niveau.
 - On utilise la primitive **let**

IV. Les Fonctions

■ Primitive let

- **let** permet de déclarer les variable locales
- **Syntaxe**

```
(let ([var1 exp1]  
      ...  
      [varn expn]  
      )  
  corps  
)
```

```
(let ((var1 exp1)  
      ...  
      (varn expn)  
      )  
  corps  
)
```

- On peut remplacer le crochets par des accolades
- Cette expression évalue les valeurs expi de gauche à droite, crée un nouvel emplacement pour chaque identifiant et place les valeurs dans les emplacements.
- Il évalue ensuite les corps dans lesquels les identifiants sont liés.

IV. Les Fonctions

□ Exemple 1 :

```
(let ([x 5])  
  x  
)  
  
5
```

- On crée la variable x et on l'initialise avec 5

□ Exemple 2 :

```
(let ((x 1) (y 5))  
  (+ x y)  
)  
  
6
```

- On crée les variable x et y on les initialise, puis on fait leur somme

IV. Les Fonctions

□ Exemple 3 :

```
(let ([x 5])  
  (let ([x 2] [y x])  
    (list y x)  
  )  
)  
'(5 2)
```

```
(let ((x 5))  
  (let ((x 2) (y x))  
    (list y x)  
  )  
)  
'(5 2)
```

- Le résultat du deuxième let devient le corps du premier let
- L'évaluation s'effectue de la gauche vers la droite, [x 2] puis [y x]. Donc x prend valeur 2, ensuite y prend la valeur initiale de x 5.
- **Pour que y puisse prendre de la valeur de x définie dans le corps de premier let, il utiliser let* (cf. plus loin).**
- On obtient donc la liste (5 2) qui sera affiché comme résultat

IV. Les Fonctions

- **Exercice :**
 - **résolution d'une équation du second degré**
 - **$ax^2 + bx + c = 0$ avec $a \neq 0$**

IV. Les Fonctions

■ **let***

- **Similaire à let, mais évalue les exp_i un par un, en créant un emplacement pour chaque identifiant dès que la valeur est disponible.**
- **Les identifiants sont liés dans les expr_i restants ainsi que dans les corps, et les identifiants ne sont pas nécessairement distincts.**
- **les liaisons ultérieures masquent les liaisons antérieures.**
- **Exemple :**

```
(let* ([x 1] [y (+ x 1)])  
      (list y x))  
  
'(2 1)
```

- Ici dès que x est initialisé, on peut l'utiliser pour initialiser y
- Ce qui n'était pas avec let de l'exemple 3 sur let.

IV. Les Fonctions

- **Exemple 2** : Exemple 3 sur let.

```
(let ([x 5])  
  (let* ([x 2] [y x])  
    (list y x)  
  )  
)  
'(2 2)
```

IV. Les Fonctions

- **4.4. Fonctions récursives**
 - **Fonction qui s'appelle elle-même**
 - **Exemple 1 : somme de deux entiers**

Algorithme	Scheme
<pre>FONCTION Somme(n m) DEBUT Si m = 0 Alors retourner n Sinon retourner 1 + somme(n, m -1) FinSi Fin</pre>	<pre>(define (somme n m) (if (= m 0) n (+ 1 (somme n (- m 1))))) > (somme 2 4) 6</pre>

IV. Les Fonctions

■ Exemple 2 :

- Définition de la fonction minimum(L) qui retourne le minimum d'une liste.

Algorithme	Scheme
<pre>FONCTION minimum(liste) DEBUT Si vide(reste(L)) Alors retourner premier(L) Sinon Si premier(L) < minimum(reste(L)) Alors retourner premier(L) Sinon retourner minimum(reste(L)) FinSi FinSi Fin</pre>	<pre>(define minimum (lambda (liste) (if (null? (cdr liste)) (car liste) (if (< (car liste) (minimum (cdr liste))) (car liste) (minimum (cdr liste))))) > (minimum '(3 2 5 1 8)) 1</pre>

IV. Les Fonctions

■ Exercices

- **Exercice 1 : écrire un programme pour $n!$**
- **Exercice 2 : Ecrire une programme permettant de calculer a^n .**
- **Exercice 5 : Ecrire une fonction qui calcule le pgcd de deux entiers a et b en utilisant l'algorithme d'Euclide :**
 - Si $a > b$, $\text{pgcd}(a, b) = \text{pgcd}(a-b, b)$
 - Si $b > a$, $\text{pgcd}(a, b) = \text{pgcd}(a, b-a)$
 - Si $a = b$, $\text{pgcd}(a, b) = a$
- **Exercice 6 : Suite de Fibonacci**
 - $U_0 = 0$
 - $U_1 = 1$
 - $U_n = U_{n-1} + U_{n-2}$

IV. Les Fonctions

□ Explication de la suite de Fibonacci

■ Reproduction des lapins :

- **Mathématicien italien Leonardo de Pise, surnommé Fibonacci**
- **« Possédant au départ un couple de lapins , combien de couples de lapins obtient-on en 12 mois si chaque couple engendre tous les mois un nouveau couple à compter du 2 mois de son existence? »**

Janvier: 1 couple
Février: 1 couple
Mars: $1 + 1 = 2$ couples
Avril: $2 + 1 = 3$ couples
Mai: $3 + 2 = 5$ couples
Juin: $5 + 3 = 8$ couples
Juillet: $8 + 5 = 13$ couples
Août: $13 + 8 = 21$ couples
Septembre: $21 + 13 = 34$ couples
Octobre: $34 + 21 = 55$ couples
Novembre: $55 + 34 = 89$ couples
Décembre: $89 + 55 = 144$ couples
Janvier: $144 + 89 = 233$ couples

à l'exception des 2 premiers, chaque terme de la suite est égal à la somme des 2 termes qui le précèdent immédiatement
Autrement il s'agit d'une suite de nombres dans laquelle tout nombre (à partir du troisième) est égal à la somme des deux précédents.

IV. Les Fonctions

□ 4.6. Fonction paramètre d'une autre fonction

- Le meilleur exemple de fonction passée en paramètre est la fonction appliquer qui prend en paramètres une fonction et une liste et qui rend la liste des éléments résultats de l'application de la fonction aux éléments de la liste.

- Exemple :

- Appliquer (+ 1 (1 2 3)) = (2 3 4)

```
(define appliquer (lambda (nomF liste)
  (cond
    ((null? liste) '())
    (else (cons (nomF (car liste)) (appliquer nomF (cdr liste))
                )
            )
  )
)
```

```
(define plusun (lambda (x)
  (+ 1 x)
)
```

```
(> (appliquer plusun (list 1 2 3))
'(2 3 4))
```

IV. Les Fonctions

■ Remarque :

- En Scheme, la fonction appliquer est une fonction prédéfinie appelée map

```
> (map plusun (list 1 2 3))  
'(2 3 4)
```

■ Exercice

- Définir la fonction carre qui prend une liste en entrée et rend la liste des éléments de élevés au carré.
- > (carre '(2 5 4 1))
- (4 25 16 1)

IV. Les Fonctions

▣ 4.7. Fonction en retour

V. Les structures de données

▣ Les arbres

- <http://aqualonne.free.fr/Teaching/csc/Scheme3.pdf>

```
(define (make-walker T pred ?)
  (define (explore S cont !)
    (if (pair? S)
        (explore (car S) (lambda () (explore (cdr S) cont !)))
        (if (null? S)
            (cont !)
            (if (pred? S)
                (begin (set ! glob-cont ! cont !) S)
                (cont !)
            )
        )
    )
  )
  (define (glob-cont !)
    (explore T (lambda () #f))
  )
  (lambda () (glob-cont !))
)
```

V. Les structures de données

- **Vecteurs ;**
- **paires orientées ;**
- **Listes ;**
- **listes associatives ;**
- **tables de hachage.**

VI. Les fichiers

VII. Système Expert

□ 7.1. Présentation

- **D:\Cours\Cours2019\IA\Support**
- **Un minuscule système expert, basé sur l'article « Introduction aux systèmes experts » [12] qui décrit une base de données botanique.**
- **Voici la traduction en Scheme de la base de règles, dans laquelle le symbole « - » représente la négation :**

VII. Système Expert

■ Base de connaissance

R01 fleur \wedge graine \rightarrow phanérogame
R02 phanérogame \wedge graine-nue \rightarrow sapin
R03 phanérogame \wedge 1-cotylédone \rightarrow monocotylédone
R04 phanérogame \wedge 2-cotylédone \rightarrow dicotylédone
R05 monocotylédone \wedge rhizome \rightarrow muguet
R06 dicotylédone \rightarrow anémone
R07 monocotylédone \wedge \neg rhizome \rightarrow lilas
R08 feuille \wedge fleur \rightarrow cryptogame
R09 cryptogame \wedge \neg racine \rightarrow mousse
R10 cryptogame \wedge racine \rightarrow fougère
R11 \neg feuille \wedge plante \rightarrow thallophyte
R12 thallophyte \wedge chlorophylle \rightarrow algue
R13 thallophyte \wedge \neg chlorophylle \rightarrow champignon
R14 \neg feuille \wedge \neg fleur \wedge \neg plante \rightarrow colibacille

VII. Système Expert

■ BC sous forme de liste

```
#lang racket/base
; BC
(define BC '(
  phanérogame (fleur graine)
  sapin (phanérogame graine-nue)
  monocotyledone (phanérogame 1-cotyledone)
  dicotyledone (phanérogame 2-cotyledone)
  muguet (monocotyledone rhizome)
  anemone (dicotyledone)
  lilas (monocotyledone - rhizome)
  cryptogame (feuille - fleur)
  mousse (cryptogame - racine)
  fougere (cryptogame racine)
  thallophyte (- feuille plante)
  algue (thallophyte chlorophylle)
  champignon (thallophyte - chlorophylle)
  collibacille (- feuille - fleur - plante)
))
```

□ **Remarque : - représente la négation**

VII. Système Expert

■ L'ensemble des buts :

```
(define BUTS '(  
  sapin muguet anemone lilas mousse  
  fougere algue champignon collibacille  
  )  
)
```

VII. Système Expert

■ Le moteur d'inférence

```
(define (deduire vrais faux BC BUTS)
  (define (tente truc vrais faux)
    (define (valide? proposition vrais faux)
      (or (null? proposition)
          (and (eq? '- (car proposition))
                (memq (cadr proposition) faux)
                (valide? (cddr proposition) vrais faux)
                )
          )
      (and (memq (car proposition) vrais)
            (valide? (cdr proposition) vrais faux)
            )
      )
    )
  (if (valide? (cadr truc) vrais faux)
      (car truc)
      #f
      )
  )
)
```

```
(define (balaye base vrais faux)
  (if (null? base)
      #f
      (essai (tente base vrais faux) base vrais
              faux)
      )
  )
(define (essai fait base vrais faux)
  (if (and fait (not (memq fait vrais)))
      fait
      (balaye (cddr base) vrais faux)
      )
  )
(define (itere vrais faux)
  (itere2 (balaye BDD vrais faux) vrais faux)
  )
)
```

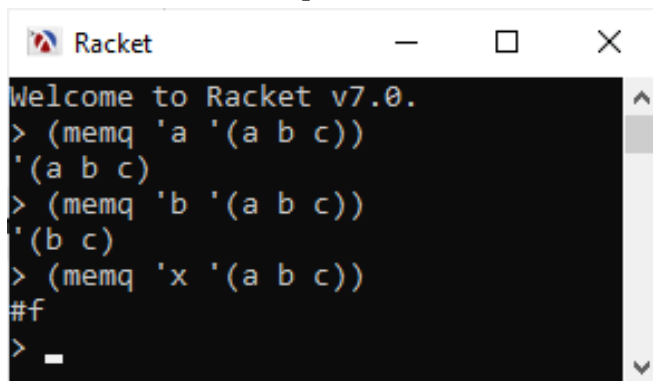
■ Vrais : BF initial

VII. Système Expert

```
(define (itere2 fait vrais faux)
  (if (not fait)
      #f
      (if (memq fait BUTS)
          fait
          (itere (cons fait vrais) faux)))
  )
)
(iterere vrais faux)
```

■ **memq : retourne une sous-liste**

□ **Exemples :**



```
Racket
Welcome to Racket v7.0.
> (memq 'a '(a b c))
'(a b c)
> (memq 'b '(a b c))
'(b c)
> (memq 'x '(a b c))
#f
> _
```

(memq 'a '(a b c))	Retourne la sous-liste commençant par a
(memq 'b '(a b c))	Retourne la sous-liste commençant par b
(memq 'x '(a b c))	x ne fait pas parti de la liste

VII. Système Expert

■ Exemple d'utilisation

- **BF initial = {rhizome fleur graine 1-cotyledone}**
- **Les conclusions que l'on peut obtenir :**
 - **(fleur graine) ⇒ phanérogame**
 - **(phanérogame 1-cotyledone) ⇒ monocotyledone**
(monocotyledone rhizome) ⇒ muguet

(deduire '(rhizome fleur graine 1-cotyledone) '() BC BUTS)
;= muguet

(deduire '(fleur graine 2-cotyledone) '() BC BUTS)
;= anemone

(deduire '(thallophyte) '(chlorophylle) BDD BUTS)
;= champignon

(deduire '(chlorophylle) '(fleur feuille rhizome) BC BUTS)
;= #f

Bibliographie

□ Bibliographie et webographie

■ Livres

■ Support de cours Web

- <http://www710.univ-lyon1.fr/~csolnon/prolog.html>
- <http://fr.foxtoo.com/install-Lisp-studio-10280948.htm>
- <http://www.ai.univ-paris8.fr/~hw/lisp/letout.pdf>
- <http://ehess.modelisationsavoirs.fr/marc/ens/deug/cours.html>