Programmation Fonctionnelle Ordre Applicatif et Formes Spéciales…

Gilles Enée – LS4

Université des Antilles Guyane

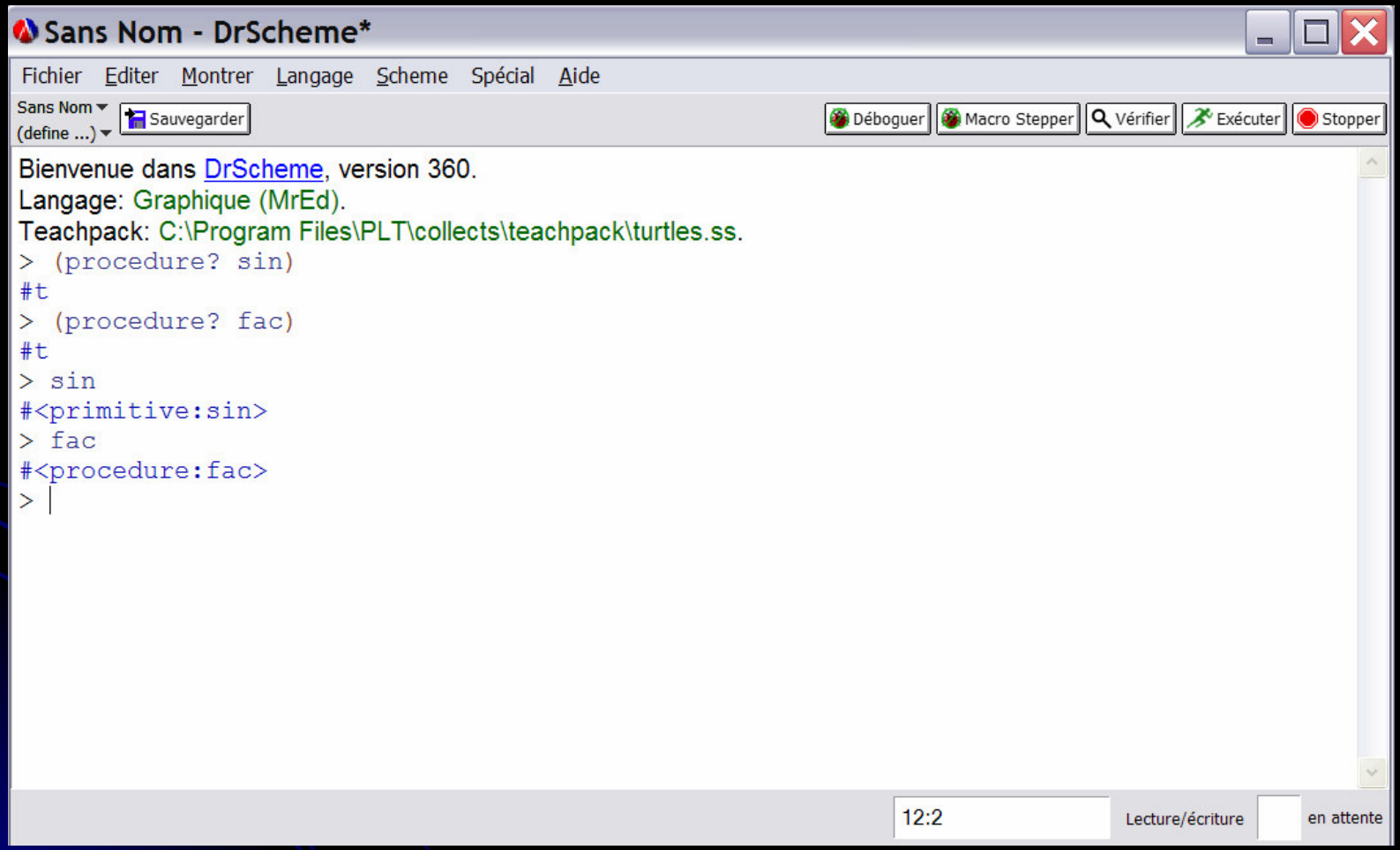
Mécanisme d'Evaluation d'un Appel de Fonction

- Supposons que l'on demande l'évaluation d'un appel de fonction de la forme :

  (f a b c)

- Où a, b et c sont des expressions quelconques.
- ATTENTION, f doit être une fonction, pas un mot-clé comme *define*, mais plutôt une primitive comme sin ou une fonction que vous avez défini.

Mécanisme d'Evaluation d'un Appel de Fonction



Mécanisme d'Evaluation d'un Appel de Fonction

- A l'aide du prédicat « procedure? » nous avons demandé si les mots clés définis étaient des procédures.
- Notez la différence entre *sin* qui est une primitive du langage et *fac* une définition faite par l'utilisateur.
- Les booléens en Scheme s'écrivent #t et #f.

Mécanisme d'Evaluation d'un Appel de Fonction

- Attention, define n'est pas un mot clé mais une forme spéciale (Testez en TP si c'est une procédure)
- Revenons à (f a b c)
  - Les quatre expressions f, a, b et c vont être évaluées [dans un ordre inconnu], donnant des valeurs F, A, B et C. La valeur F doit être un objet de type procédure sinon il y a erreur.
  - La procédure F est « appliqué » aux valeurs A, B et C pour fournir la valeur finale recherchée.

Mécanisme d'Evaluation d'un Appel de Fonction

- Vous noterez que pour décrire l'évaluation de (f a b c), nous demandons l'évaluation des sous-expressions : il s'agit donc d'une récurrence…
- Comment la procédure F est-elle appliquée ?

Mécanisme d'Evaluation d'un Appel de Fonction

- Deux possibilités :
  - Si F est une primitive comme max, elle fonctionne comme on le pense, elle calcule le maximum.
  - Si F provient de l'évaluation d'une lambda-expression, comme (lambda (x y z) (+ x (* 2 y z))), on peut imaginer que les paramètres x y et z vont être remplacés par les valeurs A, B et C, puis que l'expression (+ A (* 2 B C)) va être évaluée pour fournir la valeur recherchée.

Mécanisme d'Evaluation d'un Appel de Fonction

- Il ne s'agit que d'un modèle de « substitution » comme vous avez pu le faire en « Architecture des Ordinateurs ».
- Cet ordre de calcul se nomme « l'ordre applicatif » d'évaluation.
- En Scheme, une fonction peut aussi être calculée.
- Vous comprenez un peu mieux pourquoi *define* n'est pas une fonction :
  - Sinon en écrivant (define x 1) on chercherait à évaluer x ce qui serait absurde puisque justement on est en train de donner une valeur au symbole x.

Mécanisme d'Evaluation d'un Appel de Fonction

- Vous verrez plusieurs formes spéciales comme *define*. Chacune a un fonctionnement … spécial...

Retenez : une fonction travaille en ordre applicatif, une forme spéciale non !

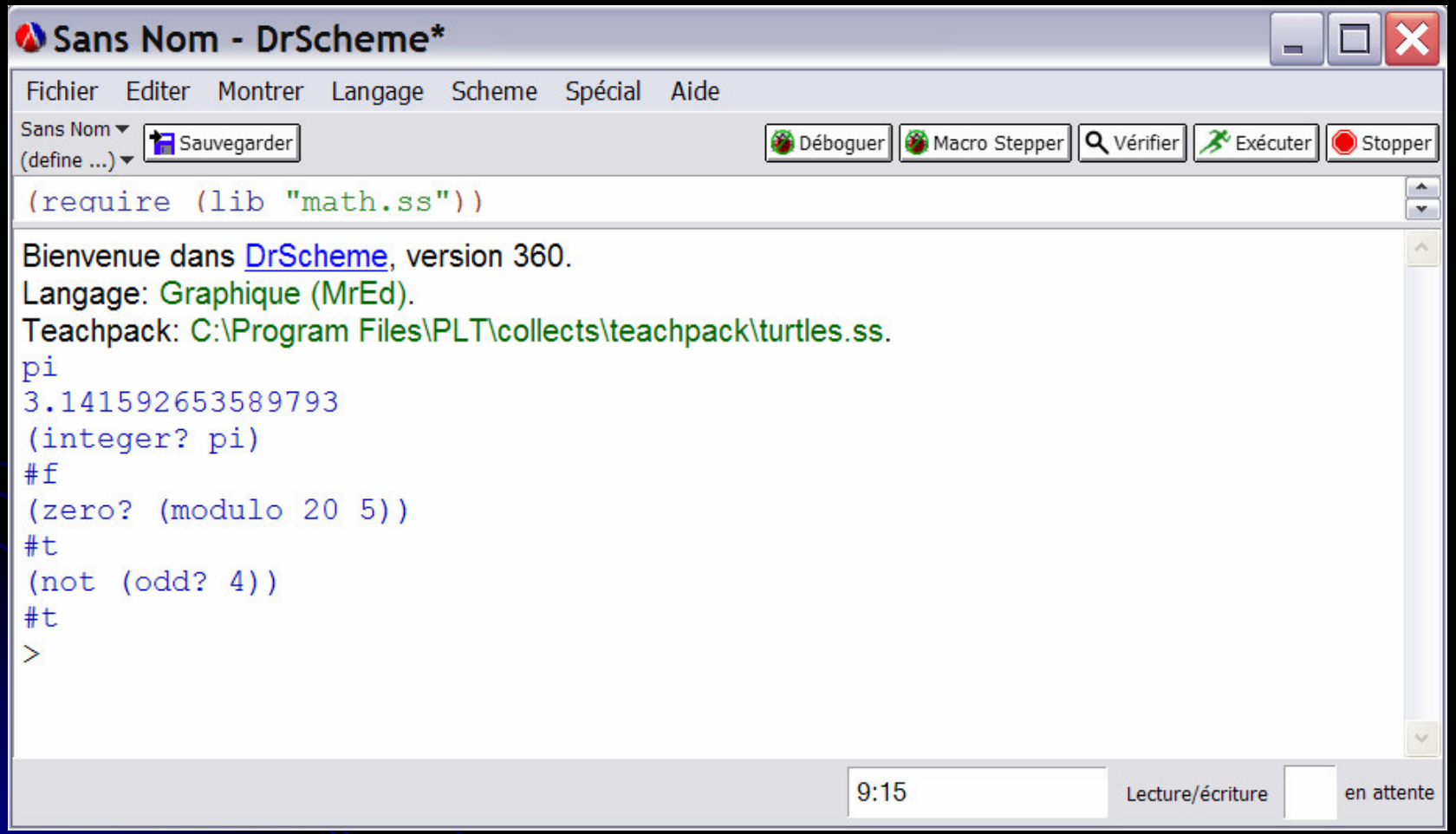
Les booléens

- Deux constantes booléennes existent :

  #t pour true et #f pour false

- On nomme **prédicat** une fonction à valeur booléenne, par exemple « integer? » (est un entier ?), « odd? » (est impair ?) ou encore « zero? » (est égal à 0 ?)

Les booléens



Les conditionnelles if et cond : Formes Spéciales

- En Scheme, nous utilisons des expression, pas des instructions. Donc la forme « if » sera une expression qui retournera un résultat, par exemple :

```
(if (integer? pi) (+ 2 3) (* 2 3))
6
```

- Le format général sera donc (if p q r) où p, q et r sont des expressions, p étant un « test ».

Les conditionnelles if et cond : Formes Spéciales

- Comment définiriez-vous (abs x) retournant la valeur absolue d'un nombre réel x si elle n'était pas définie en Scheme:

(abs -2) -> 2

N.B. Notez qu'en Scheme, vous pouvez redéfinir certaines fonctions prédéfinies (ses primitives). Ce n'est pas toujours malin, par exemple, si votre définition est mauvaise ou moins efficace que la sienne : toute liberté se paye… Notez donc votre fonction $abs plutôt qu'abs pour ne pas tuer la primitive.

Les conditionnelles if et cond : Formes Spéciales

- Il est assez désagréable d'avoir des if emboîtés, à la fois pour la lisibilité et parce que l'indentation [distance à la marge] file très vite à droite. A cet effet, Scheme dispose de la forme *cond* plus générale :

```
(cond (t1 e1)
      (t2 e2)
      (else e3))
```

```
(if t1
    e1
    (if   t2
          e2
          e3))
```

Les conditionnelles if et cond : Formes Spéciales

- Chaque t_i est un test, par exemple (> x y) et chaque e_i une expression, par exemple (+ x 1). Il peut y en avoir autant que l'on veut !
- Notez que le mot *cond* est toujours suivi de deux parenthèses (pourquoi ?)
- Il faut bien comprendre que les tests sont fait les uns après les autres. Dès qu'un test est vrai, la chaîne s'arrête et on renvoie le e_i correspondant au test t_i réussi.

Les conditionnelles if et cond : Formes Spéciales

- Comment définiriez-vous abs avec cond ?
- Fort de cet exercice, supposons que l'on veuille simuler le *if* de Scheme en utilisant *cond*. Montrez que la définition suivant est incorrecte en donnant un contre-exemple où (if p q r) et ($if p q r) ne fonctionnerait pas de la même façon :

```
(define ($if p q r) ; à tester ($if (integer? pi) …)
   (cond      (p q)
              (else r)))
```

Les conditionnelles and et or sont aussi des Formes Spéciales !

- *and* et *or* ne sont pas des fonctions à valeurs booléennes en Scheme !
- Elles ne sont pas commutatives.
- Par exemple dans (and #f q), il est parfaitement inutile d'aller perdre du temps à évaluer le test q puisque de toute façon le résultat sera faux !

Les conditionnelles and et or sont aussi des Formes Spéciales !

- Donc sous le and se cache un if :

(and p q)	(if p q #f)
(and p q r)	(if p (and q r) #f) ; Capiché ? ☺

- Autrement dit, on cherche le premier test donnant #f. Dès que l'on en trouve un, on sort du and avec comme résultat #f.
- Sinon, le résultat est celui du dernier test.

Les conditionnelles and et or sont aussi des Formes Spéciales !

- On peut donc écrire par exemple :

(and (rational? obj) (= (numerator obj) 2))

- Le numérateur d'obj ne sera considéré que si et seulement si obj est bien un rationnel.
- C'est la même chose qu'en C en réalité.
- Exercice : Que vont renvoyer

(and (< pi 3) (integer? (/ 0)))

(and (integer? (/ 0)) (< pi 3))

Les conditionnelles and et or sont aussi des Formes Spéciales !

- Sous le or se cache aussi un if :

(or p q)	(if p p q) ; p calculé 1 fois !
(or p q r)	(if p p (or q r))

- La forme or est duale : un or est vrai si au moins l'un des tests est vrai.
- On cherche donc le premier qui donne une valeur vraie et on rend cette valeur.
- Sinon on rend la valeur du dernier.

Les Variables Locales et la Forme let

- En maths, vous êtes habitués à faire des « changements de variables ».
- Par exemple, soit à calculer la fonction :

$$f(x,y) = x^2 + y^3 + \sin(x^2 - y^3)$$

- On ne souhaite pas calculer deux fois ni x^2, ni y^3. Le mathématicien va raisonner en termes de fonctions composées, et remarquer qu'en réalité :

$f(x,y) = g(x^2, y^3)$ avec $g(u,v) = u+v + \sin(u-v)$

Les Variables Locales et la Forme let

- En Scheme, cela revient à introduire une fonction auxiliaire. Vérifiez que x^2 et y^3 ne sont calculés qu'une fois :

```
(define (f x y)
   (g (* x x) (* y y y)))
```

```
(define (g u v)
   (+ u v (sin (- u v))))
```

Les Variables Locales et la Forme let

- L'aire A d'un triangle quelconque de côté a, b, c est donnée par la formule :

$$A = \sqrt{(p(p-a)(p-b)(p-c))}$$

- Où p représente le demi-périmètre. Ecrire la fonction (aire a b c) retournant l'aire d'un triangle, mais en ne calculant le demi-périmètre qu'une seule fois et sans utiliser *let* pour les petits malins qui se seraient déjà renseignés dessus.

Les Variables Locales et la Forme let

- Le changement de variables $u=x^2$ et $v=y^3$ était en fait possible en une seule fonction par la forme *let* qui permet de déclarer et d'initialiser plusieurs variables locales à un calcul. En reprenant l'exemple de f(x,y) :

```
(define (f x y) ; attention aux parenthèses !
   (let ((u (* x x)) (v (* y y y)))
        (+ u v (sin (- u v)))))
```

Les Variables Locales et la Forme let

- Forme générale de let :

(let ((x_1 e_1) (x_2 e_2) …)
<expr>)

- Où x_1, x_2, … sont des symboles et e_1, e_2, … des expressions.
- Les expressions e_i sont toutes calculées « en même temps » dans le contexte courant, puis les valeurs obtenues deviennent temporairement les valeurs des variables x_i, le temps de calculer la valeur de <expr> qui forme le « corps » du let.
- Une fois cette valeur calculée, les variables x_i reprennent leurs anciennes valeurs [si elles en avaient].

Les Variables Locales et la Forme let

- Refaites l'exercice sur l'aire du triangle avec un let.
- Sachant que (define x 2) a été fait au toplevel. Prévoyez ce que valent les expressions suivantes :

```
(let ((y (+ x 1)) (x (* x 2)))
   (+ x y))
(let ((x (* x 2)) (y (+ x 1))
   (+ x y))
```

Les Variables Locales et la Forme let

- Que pensez-vous de la version suivante de la fonction $f(x, y) = x^2+y^3$:

```
(define (f x y)
   (let ((u (* x x)) (w (* y y)) (v (* w y)))
        (+ u v (sin (- u v)))))
; A essayer en TP pour voir la réponse de Scheme
```

Les Variables Locales et la Forme let

- La définition précédente fonctionnerait par contre en remplaçant let par *let** qui force les liaisons des variables locales de la gauche vers la droite.
- Cela permet donc pour v d'utiliser la valeur que vient de prendre w, ce que ne permet pas let. [let reste plus rapide]
- A tester en TP avec let*

Les Variables Locales et la Forme let

- Comment pourriez-vous poser v = wx si let* n'existait pas ? [En réalité sous let* se cache let…]