1 Définitions et fonctions

1.1 Introduction

Racket [2] est un langage fonctionnel [3] ; les atouts des langages fonctionnels [4] sont de faciliter :

— le raisonnement et l'expression dans les programmes ;
— la compréhension des programmes.

2. S'écrit Racket et se prononce raquette.

3. Un grand nombre d'informaticiens voit la programmation fonctionnelle comme l'essence de la programmation ; dans les années 1930, avant l'existance des ordinateurs, le mathématicien Alonzo Church développa un langage (ou système formel) appelé lambda calcul permettant de définir les notions de fonction et d'application sous forme de logique combinatoire ; avec le lambda-calcul, Alan Turing développa les machines de Turing et la théorie de la calculabilité ; en 1958, John Mac Carthy créa le langage Lisp et réalisa ainsi la première solution pratique liant un langage de programmation et le lambda calcul ; dans les années 1970, apparaissent Lisp Machine Lisp, Scheme et NIL (New Implementation of Lisp) ; Gerald Jay Sussman et Guy Lewis Steele créent Scheme (à prononcer « skiim ») dans le but de simplifier Lisp avec une syntaxe extrèmement simple et peu de mots-clés ; Common Lisp apparait dans les années 80 avec la volonté inverse d'avoir des procédures et des types complexes ; Racket est une évolution de Scheme, dévelopée par le groupe PLT (Programming Language Team) de Rice University ; DrRacket (à prononcer « docteur racket ») est un environnement de développement permettant de programmer en Racket. Les références concernant le langage Racket sont `https://racket-lang.org/` et `https://docs.racket-lang.org/`.

4. Comparativement aux langages impératifs et objets, l'utilisation d'un langage fonctionnel implique pour les programmeurs une sémantique plus stable ; c'est à dire que pour un même problème, selon deux programmeurs, les programmes fonctionnels solution de ce problème auront des formes plus proches que leurs homologues impératifs par exemple ; cette qualité de stabilité sémantique, probablement conséquence des possibilités d'expression des langages fonctionnels, peut également devenir source de blocage pour certains programmeurs dans certains cas, nous relayant à une présentation plus formelle et plus systématique permettant de faire la preuve de sa bonne exécution, avec des fonctions conformes à la notion de fonction en mathématiques, immuable dans le temps, indépendante de l'état courant ; la programmation fonctionnelle 1) implique une écriture fonctionnelle, autrement dit, implique une programmation par application de fonctions par opposition à l'application de séquences d'instructions, 2) possède une transparence référentielle avec des développements par remplacement d'expressions sans modification de comportement, 3) permet d'assurer l'absence d'effets de bords et de fait l'absence de débordements en mémoire ; à cela, Racket possède en plus un typage dynamique, l'utilisation de références dès que possible et l'utilisation permanente d'un *garbage collector* ; le typage dynamique définit le type des variables à l'exécution, ce qui a pour attrait de simplifier la phase d'écriture des programmes et de faciliter le changement de type ; la contre-partie implique une consommation de mémoire plus importante à l'exécution (résultant de l'encodage des types lors de l'exécution), une détection des erreurs à l'exécution uniquement, une perte de rapidité d'exécution (résultant d'indirections supplémentaires), une perte de lisibilité (résultant de la suppression des types dans l'écriture des programmes) ; Racket permettant aux programmeurs qui le désirent de définir des types, il est possible de ne pas souffrir de ces deux derniers points négatifs et d'obtenir ainsi une meilleure garantie d'exécution sans erreur ; les références sont utilisées dès que possible, lors des passages de paramètres et des retours de fonctions ; au travers de manipulation d'objets et de références sur des objets, Racket permet ainsi de sortir de la programmation purement fonctionnelle et d'exploiter les atouts de la programmation objet ; le *garbage collector*, également appelé *ramasse-miettes*, permet au programmeur de ne pas gérer l'allocation de la mémoire ; les zones mémoires inutiles sont désallouées automatiquement quand l'interpréteur le juge nécéssaire ; les lacunes des langages fonctionnels résident dans une lenteur d'exécution (dont OCaml et Haskell, bien que admis comme rapides, sont au moins deux fois plus lents que des programmes C/C++ ; le lecteur interessé par des solutions rivalisants avec les programmes C/C++ pourra s'orienter vers la version CHIKEN de Scheme `http://wiki.call-cc.org/` ou encore vers la version Bigloo de Scheme développée par l'Inria Sophia-Antipolis), un déficit de librairies scientifiques (par comparaison aux langages C/C++, Python et Fortran) et de fait associé à une communauté plus modeste.

1.2 Interface DrRacket

DrRacket est séparé en deux parties ; la partie du haut est la fenêtre de définitions et la partie du bas est la fenêtre d'interaction ; classiquement, il s'agit d'écrire des programmes dans la partie haute et de regarder leur exécution dans la partie basse de l'interface présentée par DrRacket (figure 1).

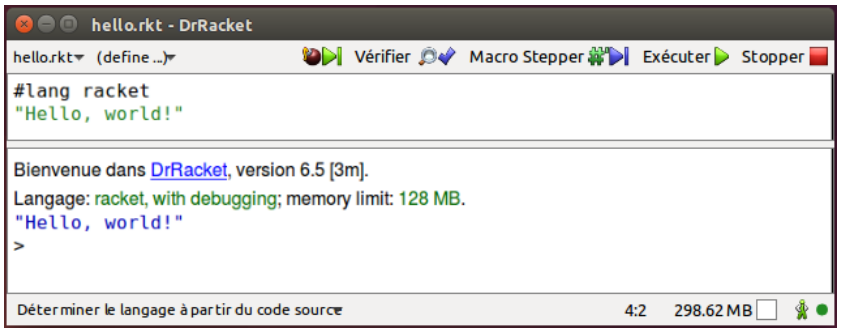

Fig. 1 – Premier programme, première exécution avec DrRacket.

Les programmes sont présentés comme suit dans ce document ; la partie supérieure présente le programme ; les lignes des programmes sont numérotées ; la partie inférieure présente l'exécution ; dans certains cas, l'exécution n'est pas présentée ; la première ligne (« `#lang racket` ») sera omise dans les programmes suivants.

```
1 #lang racket
2 "Hello, world!"
```

```
"Hello, world!"
```

1.3 Nombres primitifs, littéraux et entiers

Les principaux types de nombre sont entiers, rationnels [5], complexes, décimaux (ou flotants simple ou double précision) [6], hexadécimaux (nombres en base 16), octodécimaux (nombres en base 8) et binaires (nombres en base 2) ; certains nombres ne conservent pas leur forme initiale : les fractions sont réduites, les nombres en notation scientifique voient leur décimale déplacée pour réduire le nombre de zéros et les nombres hexadécimaux, octodécimaux et binaires sont convertis en base 10 ; le programme ci-dessous retourne `1/2` pour `2/4`, `6.02e+21` pour `0.0602e+24` et 41 pour $(29)_{16}$, 21 pour $(32)_8$ et 21 pour $(010101)_2$ en utilisant les préfixes `#x`, `#o` et `#b` [7].

5. La représentation des nombres rationnels implique l'expression du symbole `/` dans les résultats ; avec des nombres rationnels, le résultat de `1/2` est `1/2`, qui est à ne pas confondre avec `(/ 1 2)` dont le résultat est `0.5` ; `1/2` et `(/ 1 2)` sont du type `exact` et `0.5` est du type `inexact` ; étant de types différents, les comparer avec `equal?` implique une conversion avec `exact->inexact` ou `inexact->exact` ; les comparer avec `=` n'implique pas de conversion.

6. La virgule est représentée par un « `.` » selon les standards internationaux.

7. Il existe également deux autres préfixes : `#e` pour les nombres exactes et `#i` pour les nombres inexactes ; classiquement `(exact? 1+0i)` retourne `#t` alors que `(exact? #i1+0i)` retourne `#f`.

```
1
2/4
1+2i
3.14
0.0602e+23
#x29
#o32
#b010101
```

Les nombres peuvent être entiers (dans $\mathbb{Z}$), rationnels (dans $\mathbb{Q}$), décimaux (dans $\mathbb{D}$) ou encore complexes (dans $\mathbb{C}$) ; avec Racket, ils sont soient `exacts` (entiers et rationnels) soient `inexacts` (décimaux et complexes).

1.4 Opérations simples

La notation est dite préfixée ; l'opérateur précède les arguments ; un « `;` » annonce des commentaires ; commencer par « `;; ...` » permet de mieux les distinguer.

```
;; ... 1+2*3/4+5*6 s'ecrit usuellement 1+((2*3)/4)+(5*6)
(+ 1 (+ (/ (* 2 3) 4) (* 5 6)))
;; ... ou encore usuellement 1+(2*(3/4))+(5*6)
(+ 1 (+ (* 2 (/ 3 4)) (* 5 6)))
```

Dans les opérations classiques, nous avons des habitudes de lecture des priorités des opérations, correspondant à des parenthèses ; le principe de l'écriture préfixée est de noter les opérateurs devant les arguments [8] ; pour accéder à l'aide d'une fonction dans DrRacket, ou aux fonctions commençant par un nom, il suffit de presser F1 après avoir tapé le nom considéré.

8. L'écriture préfixée est une proposition faite en 1924 par le mathématicien Polonais Jan Łukasiewicz, et est de fait également appelée notation polonaise ; cette notation est justifiée par son évaluation dans un ordinateur, en association avec une pile qui permet une évaluation concomitante à sa lecture ; la sémantique de Racket se voit simplifiée, organisée par l'ajout de parenthèses permettant de distinguer fonctions et paramètres et de délimiter la portée dans les expressions ; ce choix de notation ne fait pas l'unanimité des langages fonctionnels ; c'est par exemple le cas du langage fonctionnel Erlang, créé en 1986 par la société du domaine des télécommunications Ericsson, à la syntaxe riche, à la tendance modulaire et favorable à l'expression de la concurrence dans les programmes, nécessite trois éléments « . », « ; » et « , » permettant respectivement de définir la fin, de séparer les clauses et de déparer les éléments d'une liste ; ce choix de notation sera possible sans être obligatoire avec le langage de programmation Haskell, nommé ainsi en référence au logicien Haskell Curry, dont l'objectif annoncé était à sa création en 1990, de concentrer les efforts de recherche des langages fonctionnels pratiquant l'évaluation paresseuse (*i.e.* l'évaluation à l'utilisation). Haskell possède lui aussi une syntaxe riche, résultant de l'utilisation spécifique de caractères tels que « : » pour spécifier des modes de calcul via des commandes et « : : » pour spécifier un type.

```
1  (+ 2 4) (- 2 4) (* 2 4) (/ 2 4)
2  (/ (+ 2 4) 3)
3  (expt 3 3) (sqrt 3)
4  (quotient -13 3) (remainder -13 3) (modulo -13 3)
5  (add1 4) (sub1 4)
6  (max 3 5) (min 3 5)
7  (gcd 4 6) (gcd 1/2 1/3) (lcm 3 4)
8  (round 2.6) (floor 2.6)
9  (ceiling 17/4) (ceiling -17/4) (truncate 17/4)
10 (numerator 17/4) (denominator 17/4)
11 (random 10)
12 (random)
```

La première ligne présente une suite d'appels de fonctions ; pour des raisons de commodité, les séquences d'appels sont présentées en lignes et en colonnes ; la fonction `(+ ...)` appelée ligne 1 retourne l'addition des valeurs passées en paramètre ; la fonction `(* ...)` appelée ligne 1 retourne la multiplication des valeurs passées en paramètre ; la fonction `(/ ...)` appelée ligne 1 retourne la division des valeurs passées en paramètre ; la ligne 2 présente la division du résultat d'une addition [9] ; à la ligne 3, la fonction `expt` retourne la puissance du premier nombre élevé au second [10] (içi égale à 3^3) et la fonction `sqrt` retourne la racine-carrée [11] du paramètre ; la fonction `quotient` appelée ligne 4 retourne le quotient de la division entière du premier paramètre par le second (içi égal à `-4`) ; la fonction `remainder` appelée ligne 4 retourne le reste de la division entière (içi égal à `-1`) ; la fonction `modulo` appelée ligne 4 retourne le premier paramètre modulo [12] le second (içi égal à `2`) ; la fonction `add1` appelée ligne 5 ajoute un à la valeur passée en paramètre ; la fonction `sub1` appelée ligne 5 soustrait un à la valeur passée en paramètre ; la fonction `max` appelée ligne 6 retourne la plus grande [13] des valeurs passées en paramètre (içi égal à `5`) ; la fonction `min` appelée ligne 6 retourne la plus petite [14] des valeurs passées en paramètre (içi égal à `3`) ; la fonction `gcd` appelée ligne 7 retourne le plus grand commun diviseur (*i.e pgcd*), soit 2 puis 1/6 ; la fonction `lcm` appelée ligne 7 retourne le plus petit commun multiple (*i.e. ppcm*), soit 12 ; le quadruplet de fonctions `round`, `floor`, `ceiling` et `truncate` produit des arrondis, dans l'ordre 3.0, 2.0, 5, -4 et 4 (`ceiling` étant appelée deux fois) ; le résultat est donc toujours un entier ; les fonctions `numerator` et `denominator` retourne numérateur et dénominateur des fractions en paramètre ; la fonction `random` avec 10 pour argument retourne un

9. Notons l'arité multiple des opérateurs `+`, `-`, `*` et `/` permet d'écrire des suites de valeurs et permet d'avoir des comportements différents ; l'utilisation de ces opérateurs avec plus de deux paramètres est potentiellement source de confusion et est déconseillée ; ces opérateurs pour les suites `(1 2 3 4)` retournent respectivement `10`, `-8`, `24` et `1/24`.

10. La fonction puissance est usuellement appelée `pow` ; les différences de comportement de `(expt x y)` avec `pow (x, y)` selon la spécification `C99` sur les valeurs négatives de `x` et non-entières de `y` semblent avoir poussé les développeurs de racket à nommer cette fonction `expt` et non `pow`.

11. Dans le sens où la valeur retournée est représentée dans un flottant, la valeur retournée est dans certains cas une valeur approchée de la racine-carrée (comme $\sqrt{2}$ et $\sqrt{3}$).

12. Les fonctions `(remainder x y)` et `(modulo x y)` donnent des résultats similaires pour des valeurs positives et des résultats différents pour des valeurs négatives ; le `remainder` étant toujours un reste, donc de même signe que la première valeur `x` tandis que le `modulo` retourne une valeur variant de 0 à `y-1`, donc de même signe que `y`.

13. La plus grande valeur est couramment appelée le maximum et abrégé le max.

14. La plus petite valeur est couramment appelée le minimum et abrégé le min.

entier dans [15] l'ensemble $\{0, 1, \ldots, 9\}$; sans argument, la fonction `random` sans argument retourne un nombre réel dans $[0; 1]$.

1.5 Prédicats

Les prédicats sont des fonctions qui retournent vrai ou faux ; `#t` abrège « true » pour dire vrai ; `#f` abrège « false » pour dire faux [16].

```
(not #t)
(even? 1)
(odd? 2)
(integer? 1.1)
(rational? 1.1)
(real? 1.1)
(complex? -1)
(boolean? 1)
(number? 'a)
(negative? 1)
(positive? -1)
(zero? 1)
(exact-integer? 2.2)
(exact-nonnegative-integer? -1)
```

Toutes les évaluations des prédicats précédents retournent faux ; en commençant par `not` qui est la négation, puis `even` pour est-pair, puis `odd` pour est-impair, en finissant par `exact-integer` pour est-entier et par `exact-nonnegative-integer` pour est-entier-strictement-positif ; le caractère `?` souligne la spécificité de leur retour qui est vrai ou faux ; ces prédicats assurent, en cas de réponse vrai, une possibilité de conversion dans le type associé ; dans ce sens, `(integer? 1.0)` retourne vrai et `rational?` indique si un nombre est non imaginaire [17] ; dans certains cas, la valeur d'une variable est un nombre sans valeur numérique ou un nombre avec une valeur infinie [18].

Les opérateurs de comparaison numérique impliquent que leur opérandes soient numériques.

```
(= 2 4) (< 1 1) (<= 2 1) (>= 1 2)
```

Toutes les évaluations des prédicats précédents retournent encore faux.

15. Notons qu'appeler `(random 1)` retourne toujours `0`.

16. `#T` est équivalent à `#t` et `#F` est équivalent à `#f` mais l'écriture en minuscules est préférée ; ce qui explique pourquoi les expressions `(boolean? #t)`, `(boolean? #f)`, `(boolean? #T)` et `(boolean? #F)` sont toutes vraies et les expressions `(boolean? "yes")` et `(boolean? "no")` sont fausses.

17. Ce qui explique que : `(rational? (sqrt 2))` retourne vrai, `(sqrt 2)` étant une approximation rationnelle de $\sqrt{2}$; `(rational? (sqrt -1))` retourne faux, la racine d'un nombre négatif étant un nombre complexe.

18. Pour un nombre, être sans valeur numérique équivaut à ne pas être un nombre au sens classique, ou encore équivaut à avoir pour valeur `+nan.0` (un nombre positif sans valeur numérique) ou `-nan.0` (un nombre négatif sans valeur numérique) ; le prédicat `nan?` permet de tester le fait d'être sans valeur numérique ; les valeurs infinies sont abrégées `+inf.0` et `-inf.0` ; `(/ 1 +inf.0)` retourne `0.0`, `(/ 1 -inf.0)` retourne `-0.0`, `(max 34 -inf.0)` retourne 34 et `(min 34 +inf.0)` retourne 34 ; dans le cas de Racket, aucune fonction prédéfinie ne retourne `nan` et `inf` ; dans le cas de Emacs Lisp, la division d'une valeur float par zéro retourne `nan`.

1.6 Définition et test : `define`, `lambda` et `if`

La fonction `define` permet de définir des constantes et des fonctions qu'il sera possible de réutiliser plus tard.

```
1 (define A 10)
2 A
```

```
10
```

`A` est une constante égale à 10 (définie ligne 1 et appelée/affichée ligne 2).

```
1 (define (f x) (if (>= x 10) #t #f))
2 (f 9) (f 10) (f 11)
```

```
#f
#t
#t
```

On définit (ligne 1) une fonction `f` « est supérieure ou égale à 10 » ; ensuite on appelle/utilise `f` (ligne 2) pour calculer les images des valeurs 9, 10 et 11.

La définition de la fonction `f` est différente de la définition précédente de la constante `A` dans le sens ou elle définit une fonction ; cette particularité est mise en avant dans la formulation utilisant le mot clé `lambda` ; la formulation précédente de `f` reste équivalente.

```
1 (define f (lambda (x) (if (>= x 10) #t #f)))
2 (f 9) (f 10) (f 11)
```

```
#f
#t
#t
```

Il est également possible d'utiliser `lambda` en utilisant et définissant *in situ* les traitements recherchés (correspondant aux fonctions classiquement définies, içi `(f 9)`).

```
1 ((lambda (x) (if (>= x 10) #t #f)) 9)
```

```
#f
```

Définissant des fonctions (nécessitant parfois d'autres définitions de fonction), il apparaît les notions de fonction principale et auxiliaire [19] ; dans le programme ci-dessous, la fonction `g1` est la fonction principale qui utilise la fonction auxiliaire `f`, déclarée pour cette occasion ; afin de restreindre la déclaration d'une fonction, on peut encapsuler un nouveau `define` comme dans les définitions de `g2` et `g3` (équivalentes à la fonction `g1`) ; dans le cas de `g2` et de `g3`, les fonctions `h` sont locales aux définitions ; l'encapsulation de `define` évite une multiplication des fonctions principales qui, dans certains cas, nuira à la bonne compréhension des programmes.

```
1  (define (f x y) (* x y))
2  (define (g1 x) (f x x))
3  (define (g2 x)
4    (define (h x y) (* x y))
5    (h x x))
6  (define (g3 x)
7    (define h (lambda (x y) (* x y)))
8    (h x x))
9  (g1 8)
10 (g2 16)
11 (g3 32)
```

```
64
256
1024
```

Le test conditionnel « `if` » permet de sélectionner une expression à exécuter ; le test `if` se compose d'une condition et de deux expressions.

```
1  (define (f x) (if (> x 10) (+ x 1) (- x 1)))
2  (f 5) (f 15)
```

```
4
16
```

Si la condition est vérifiée, la première expression est exécutée ; sinon la deuxième expression est exécutée [20].

19. La fonction principale est également appelée fonction chapeau de la fonction auxiliaire.

20. Contrairement à de nombreux langages de programmation, la deuxième expression `else` est impérativement présente, ce qui permet d'éviter d'éventuelles ambiguïtés ; la présence obligatoire du `else` vient du fait que le `if` est une fonction et doit de fait retourner quelquechose (autrement dit, ne peut pas rien retourner dans les cas `else`) ; imposer un retour implique donc d'imposer la présence d'un `else` pour chaque `if` dans le programme ; il en est de même pour la fonction `cond` ; à l'inverse, il est possible en langage `C` de ne rien retourner en utilisant `void`, qui est à la fois rien dans ce cas et à la fois tout dans le cas de `void*`, qui est une adresse par définition ; de la même façon, il est possible en Racket de retourner une constante `(void )` pour ne rien faire dans la deuxième partie d'une fonction ; cette solution n'est cependant pas recommandée.

Pour une succession de tests `if`, il est possible d'utiliser `cond` qui présente une énumération de tests, énumération se terminant le cas échéant par l'exécution de la fonction associée à un `else` ; les deux fonctions `f1` et `f2` ci-dessous sont équivalentes.

```
(define (f1 x)
  (if (> x 10)
    (+ x 1)
    (if (> x 100)
      (+ x 10)
      (- x 1)
    )))
(define (f2 x)
  (cond ((> x 10) (+ x 1))
        ((> x 100) (+ x 10))
        (else (- x 1))
  ))
```

On note que les deux fonctions `f1` et `f2` possèdent la même partie de code mort ; le test `(> x 100)` étant toujours faux, la fonction `(+ x 10)` n'est jamais exécutée.
Pour plus de clarté, on pourra noter les conditions entre crochets.

```
(define (f2 x)
  (cond [(> x 10) (+ x 1)]
        [else (- x 1)]
  ))
```

Il est possible de combiner les expressions retournant des valeurs booléennes avec les opérateurs `and` et `or` correspondants respectivement au et-logique et au ou-logique.

```
(and #t #f)
(and (= 1 1) (= 2 2))
(or #f #t)
(or (= 1 2) (= 1 0))
```

```
#f
#t
#t
#f
```

Il est possible de passer des valeurs de paramètre par défaut [21].

```
(define (f [n 3]) (if (= 0 n) 0 (+ 1 (f (- n 1)))))
(f)
(f 4)
```

```
3
4
```

21. L'utilisation des crochets comme des parenthèses est possible ; écrire `(f (n 3))` est içi équivalent à `(f [n 3])` ; les crochets seront une fois de plus préférés pour des raisons de clarté.

1.7 Variables locales, `let`, `let*`, `letrec` et `local`

Il est parfois nécessaire [22] d'utiliser plusieurs fois une fonction avec les mêmes paramètres dans une expression ; pour éviter l'exécution redondante d'appels de fonctions avec des paramètres identiques, il est possible d'utiliser des variables locales, liant les expressions entre elles ; commençons par prendre pour exemple le calcul de $\sqrt{x^2+1}+\sqrt{x^2+2}$.

```
1 (define (f x) (+ (sqrt(+ 1 (* x x))) (sqrt(+ 2 (* x x)))))
```

On peut utiliser soit les fonctions `let`, `let*` et `letrec` héritées de Scheme, soit la fonction `local` de Racket.

La fonction `let` permet de définir une variable équivalente à ces appels et précise ainsi à l'interpréteur qu'il suffit de l'exécuter une seule fois et de réutiliser la ou les valeurs retournées ; les crochets seront utilisés pour séparer les variables [23].

```
1 (define (f x)
2   (let ([x2 (* x x)])
3     (+ (sqrt(+ 1 x2)) (sqrt(+ 2 x2)))
4   ))
```

Pour le calcul de $\sqrt{x^2+x^3}+\sqrt{x^2+1}+x^3$, on peut utiliser `let` pour définir deux variables.

```
1 (define (f x)
2   (let ([x2 (* x x)]
3         [x3 (* x x x)])
4     (+ (sqrt(+ x2 x3)) (sqrt(+ 1 x2)) x3)
5   ))
```

Avec `let*`, on peut définir les variables en fonction des variables précédentes.

```
1 (define (f x)
2   (let* ([x2 (* x x)]
3          [x3 (* x2 x)])
4     (+ (sqrt(+ x2 x3)) (sqrt(+ 1 x2)) x3)
5   ))
```

Avec `local`, on peut également définir des variables en fonction des variables précédentes.

```
1 (define (f x)
2   (local ([define x2 (* x x)]
3           [define x3 (* x2 x)])
4     (+ (sqrt(+ x2 x3)) (sqrt(+ 1 x2)) x3)
5   ))
```

22. On évitera tant que faire se peut l'utilisation des variables locales dans les programmes.
23. Crochets et parenthèses sont içi syntaxiquement équivalents mais les crochets facilitent la lecture.

Pour le calcul de $\sqrt{x! + x^2} + \sqrt{x! + 1}$, on peut utiliser `letrec` pour définir des fonctions définies récursivement comme des variables.

```
1  (define (f x)
2    (letrec ([fac (lambda (n) (if (= n 1) 1 (* n (fac (sub1 n)))))]
3             [x2 (* x x)])
4      (+ (sqrt(+ x2 (fac x))) (sqrt(+ 1 (fac x))))
5    ))
```

Ce qui avec `local` s'écrit légèrement différement.

```
1  (define (f x)
2    (local ([define (fac n) (if (= n 1) 1 (* n (fac (sub1 n))))]
3            [define x2 (* x x)])
4      (+ (sqrt(+ (fac x) x2)) (sqrt(+ (fac x) 2)))
5    ))
```

On note que `local` est équivalent à `let*` mais pas à `let` ; `let` déclare des variables indépendantes ; `local` et `let*` déclarent des variables liées dans l'ordre de leur déclaration ; il est donc important d'être très vigilant sur les valeurs des variables avant leur utilisation avec `let`, `let*` et `local`, sans quoi des ambiguités peuvent apparaitre.

```
1  (define x 0)
2  (let ([x 1]
3        [y (* x x)])
4    y)
5  (let* ([x 1]
6         [y (* x x)])
7    y)
8  (local ([define x 1]
9           [define y (* x x)])
10   y)
```

```
0
1
1
```

Avec `let`, la constante `x` (ligne 1) est utilisée pour le calcul de `y` (ligne 3) ; avec `let*` et `local`, les variables locales `x` (lignes 5 et 8) sont utilisées pour le calcul de `y` (lignes 6 et 9).

1.8 Trace d'exécution

En ajoutant le module `racket/trace` et en demandant la trace d'exécution d'une fonction, il est possible de voir ses appels successifs ; la fonction `trace`, prend en paramètre les fonctions dont les traces d'exécution sont désirées.

```
1 (require racket/trace)
2 (define (f [n 3]) (if (= 0 n) 0 (+ 1 (f (- n 1)))))
3 (define (g n) (if (= n 0) 10 (g (- n 1))))
4 (trace f g)
5 (f)
6 (g 3)
```

```
>(f)
> (f 2)
> >(f 1)
> > (f 0)
< < 0
< <1
< 2
<3
3
>(g 3)
>(g 2)
>(g 1)
>(g 0)
<10
10
```

Les caractères chevron ouvrant et fermant (« < » et « > ») permettent de voir respectivement l'empilement des appels de fonction et le retour des appels ; les fonctions empilées sont mémorisées et ne sont pas affichées ; la valeur finale retournée est affichée sans chevron.

La différence entre arguments et paramètres est contextuelle : une fonction est définie avec ou sans arguments et est utilisée (ou appelée) avec ou sans paramètres ; avec la notation parenthèsée, fonctions et paramètres sont différenciés par leur position dans les parenthèses : les fonctions sont toujours précédées d'une parenthèse ouvrante [24] ; l'arité d'une fonction est le nombre de paramètres présents à son utilisation ; dans l'expression, `(a b (c d (e (f a))) h)`, les fonctions sont `a`, `c`, `e` et `f`, dont les arités sont respectivement 3, 2, 1 et 1 ; les paramètres peuvent être eux-mêmes des appels de fonctions (comme dans le cas de la fonction `f`), voir même des retours de fonction (comme dans le cas d'un paramètre accumulateur).

24. Réciproquement, une parenthèse ouvrante ne précède pas toujours la définition d'une fonction ; avec le caractère d'échappement quote, une parenthèse ouvrante annonce une liste.

La compréhension d'une fonction passera parfois par son développement qui consiste à remplacer étape par étape, une expression conformément aux définitions ; ce développement correspond à la trace d'exécution d'une fonction, en remplaçant les chevrons correspondants aux empilements d'appels de fonction par les appels de fonction en question.

```
= (f)
= (f 3)
= (+ 1 (f 2))
= (+ 1 (+ 1 (f 1))
= (+ 1 (+ 1 (+ 1 (f 0))))
= (+ 1 (+ 1 (+ 1 0)))
= 3
= (g 3)
= (g 2)
= (g 1)
= (g 0)
= 10
```