Le langage Scheme : Niveau Intermédiaire II, arbres

D'après les cours de J.-P. Roy (Nice) et C. Queinnec (Paris)

I. Définir des arbres binaires d'expressions

On définie un arbre par une racine, un sous-arbre gauche et un sous-arbre droit. Ainsi :

```
(define (arbre r Ag Ad)
  (list r Ag Ad)) ; notre arbre est fait en interne par une simple liste
```

La particularité d'un arbre binaire d'expression est que ses nœuds sont des opérateurs (par exemple -, *, /…) et ses feuilles des opérandes (des nombres dans notre cas). Dans un arbre arithmétique, les feuilles sont des constantes mais dans un arbre algébrique elles peuvent être des variables, dont on ira chercher la correspondance dans une liste.

```
(define (feuille? obj)    ; la feuille d'un arbre algébrique
  (or (number? obj)       ; soit c'est un nombre
     (and (symbol? obj) (not (operateur? obj))))) ; soit une constante

(define (operateur? obj) (member obj '(+ - * /)))        ; les nœuds sont des opérateurs mathématiques

(define (fg A)
  (if (feuille? A)
     (error "pas de fils gauche pour une feuille !" A)
     (cadr A)))

(define (fd A)
  (if (feuille? A)
     (error "pas de fils droit pour une feuille !" A)
     (caddr A)))

(define (racine A)
  (if (feuille? A)
     (error "pas de racine pour une feuille !" A)
     (car A)))
```

Pour créer un arbre, dans un algorithme on doit passer par le type abstrait :

```
(define AT
  (arbre '- (arbre '* 2 (arbre '+ 'x 1)) (arbre '/ 'x 'y)))
```

Au point de vue interne, on peut parfois se permettre (en tests) de violer le type abstrait :

```
(define AT '(- (* 2 (+ x 1)) (/ x y)))
```

II. Algorithmique élémentaire sur les arbres

La hauteur d'un arbre est l'algorithme fondamental. On regarde pour chaque élément si c'est une feuille : si c'est le cas, alors il n'y a rien en dessous, on s'arrête. Sinon, la hauteur est 1 de plus que ce qui suit.

```
(define (hauteur A)
  (if (feuille? A) ; si on est une feuille, on s'arrête et on renvoie le neutre pour l'opération : 0
     0
     (+ 1 (max (hauteur (fg A)) (hauteur (fd A))))) ; sinon, c'est 1 de plus que la hauteur suivante : max !
```

Une variante de cet algorithme peut nous servir à indiquer le nombre de noeuds dans l'arbre :

```
(define (nb-op A)
  (if (feuille? A) ; si j'ai une feuille, je m'arrête je renvoie le neutre
     0
     (+ 1 (nb-op (fg A)) (nb-op (fd A)))) ; sinon, je tient un nœud de plus, et je continue
```

```
(define (k-nb-op A f) ; style CPS de l'algorithme précédent, on calcule f(nb-op(A))
  (if (feuille? A)
      (f 0)
      (k-nb-op (fg A) ; j'affirme avoir calculé le nombre de noeuds à gauche
            (lambda (ng) (k-nb-op (fd A) ; j'affirme avoir calculé le nombre de nœuds à droite
                  (lambda (nd) (f (+ 1 ng nd))))))))) ; il ne me reste plus qu'à sommer
```

Autre problème : une feuille est-elle dans un arbre ? Il faut se diffuser dans l'arbre. Quant on est sur une feuille, on regarde si c'est celle qu'on cherche. Si nous ne sommes pas sur une feuille, alors on se diffuse.

```
(define (feuille-de? x A)
  (if (feuille? A)
      (equal? x A) ; si c'est une feuille, on regarde si c'est la bonne. Sinon, pas une feuille : on se diffuse
      (or (feuille-de? x (fg A)) (feuille-de? x (fd A))))) ; or : si on a trouvé à gauche, on passe pas à droite !
```

On va maintenant s'intéresser à quelques parcours d'arbres[1] :

- préfixe : racine, fils gauche, fils droit. Par exemple f(x, y), ou toutes opérations en Scheme.
- infixe : fils gauche, racine, fils droite. Le plus courant, avec les opérations, par exemple x + y.
- postfixe : fils gauche, fils droit, racine. Machines à pile, comme une calculette HP.

Une utilité d'un parcours, par exemple préfixé, peut-être d'obtenir la liste de toutes les feuilles :

```
(define (feuillage A)
  (if (feuille? A)
      (list A) ; ce n'est pas A mais la liste réduite à A.
      (append (feuillage (fg A)) (feuillage (fd A))))) ; on a fait le feuillage, on colle le résultat (des listes !)
```

On fait passer ceci en itératif grâce au CPS :

```
(define (k-feuillage A f) ; principe du CPS, j'enveloppe le résultat par une fonction f que je passe
  (if (feuille? A)
      (f (list A)) ; on enveloppe simplement
      (k-feuillage (fg A) ; je calcule le feuillage à gauche
            (lambda (Lg) (k-feuillage (fd A) ; je calcule le feuillage à droite
                  (lambda (Ld) (f (append Lg Ld))))))))) ; et je mixe les résultats avec un append
```

Une fonction plus délicate à présent. Etant donné un une liste L qui résulte d'un parcours préfixe d'un arbre, on voudrait en tirer une liste à deux éléments A et R où A est l'arbre dont on a reconstruit les parenthèses et R les éléments qui restent après analyse.
Exemple : (+ * - x y z + u v a b c d) → A = (+ (* (- x y) z) (+ u v)) ; R = (a b c d)

Une solution possible est :

```
(define (arboriser L)
  (if (null? L)
      (list '() '()) ; si j'ai la liste vide, alors chacune des listes à renvoyer est vide
      (if (not (operateur? (car L))) ; si ce que j'ai n'est pas un opérateur, alors…
          (cons (car L) (list (cdr L))) ; alors le résultat c'est l'élément courant, et le surplus
          (let ((L1 (arboriser (cdr L)))) ; sinon, L1 est une liste de type ((op a b) (reste)).
            (let ((L2 (arboriser (cadr L1)))) ; et L2 est le reste de l'arbre L1.
              (cons (arbre (car L) (car L1) (car L2)) (cdr L2))))))) ; on a ((opL (op a b) (op' a' b')) (reste')).
```

Pour conclure, on veut un prédicat sous-arbre? afin de savoir si un arbre est inclus dans un autre.
Par exemple, 'x est inclus dans '(+ (* x 2) 3), tout comme (* x 2).

```
(define (sous-arbre? A1 A2)
  (or (equal? A1 A2) (and (not (feuille? A2)) (or (sous-arbre? A1 (fg A2)) (sous-arbre? A1 (fd A2))))))
```

III. Evaluation d'un arbre d'expression

On s'attache ici à donner le résultat du calcul d'un arbre d'expression. Dans un premier temps, il s'agit d'un arbre arithmétique : il n'y a que des constantes dans les feuilles. On applique donc les opérateurs contenus dans les nœuds sur le sous arbre gauche et le sous arbre droit, de deux façons possibles :

```
(define (valeur A)
 (if (feuille? A)
    A
    (let ((vg (valeur (fg A))) (vd (valeur (fd A))))
     (case (racine A) ; je regarde dans quel cas je suis
       ((+) (+ vg vd)) ; j'ai un + ? j'applique le +.
       ((-) (- vg vd)) ; j'ai un - ? pareil. Etc. etc.
       ((*) (* vg vd))
       ((/) (/ vg vd))
       (else (error "operateur inconnu !" (racine A)))))))
```

```
(define (valeur A)
 (if (feuille? A)
    A
    (let ((vg (valeur (fg A))) (vd (valeur (fd A))))
     ((eval (racine A)) vg vd))))
; je transforme le symbole contenu dans (racine A)
; en une procédure, grâce à l'utilisation de eval.
; Par contre, je ne m'occupe pas d'erreur, et le
; programme n'est plus portable car j'utilise dans
; le code l'interpréteur Scheme tout entier.
```

Le principe est classique en Scheme : j'affirme avoir fait certaines choses grâce à un let (hypothèse de récurrence), et à partir de là je suis dans un cas simple où il ne me reste plus qu'à faire l'opération finale, à savoir appliquer un opérateur sur deux valeurs (provenant du calcul du sous arbre gauche et droit).

A présent, on s'occupe d'arbres algébriques : la feuille peut-être une constante comme une variable. Dans ce cas, il faudra aller chercher la valeur correspondante à l'aide de la mémoire[2] fournie en paramètre.

```
(define (valeur A AL) ; la mémoire est une A-liste : nom de variable associé à une valeur
 (if (feuille? A) ; A est une feuille, c'est-à-dire une variable ou un nombre. Déterminons le cas.
    (if (number? A)
       A          ;A est un nombre, je renvoie simplement ce nombre.
       (let ((ass (assoc A AL))) ; sinon A est une variable : on prend sa valeur grâce à (assoc x AL)
        (if ass ;si assoc a réussi à trouver quelque chose, alors il nous renvoie le morceau. Ce n'est pas #f
           (cadr ass) ; à partir du morceau renvoyer, il suffit de faire un cadr pour avoir la valeur
           (error "variable inconnue !" A))));sinon, assoc n'a rien trouvé, la variable n'est pas en mémoire
    (let ((vg (valeur (fg A) AL)) (vd (valeur (fd A) AL))) ; A est un opérateur, je fais comme avant
      (case (racine A)
       ((cos) (cos vg)) ; on fait en intermédiaire : si on a un opérateur unaire, on le prend en charge
       ((sin) (sin vg))
       ((eval (racine A)) vg vd))))) ; et pour tout ce qui n'est ni cos ni sin, alors on applique bêtement
```

IV. Principes d'un simplificateur formel[3]

Le problème de la simplification est crucial dans les calculs : comment réduire les expressions à des formes de préférence 'canoniques', pour qu'il y ait unicité ?

L'évaluateur précédent pouvait réduire un arbre si tout était connu : au final il n'y avait que des constantes, soit directement, soit par correspondance en allant les prendre dans la mémoire.

Notre problème est ici de réduire le plus possible un arbre qui contient des inconnues. Certaines réductions sont évidentes : (* 0 x) entraîne immédiatement 0, tout comme (* 1 x) donne x. On sait ensuite que (/ (* 12 x) (* 4 x)) se simplifie en 3. Petit à petit, on est en mesure de simplifier de mieux en mieux.

```
(define (simplif A)      ; on défini un simplificateur formel, qui va tenter de réduire le plus possible l'arbre
 (if (feuille? A)
    A                    ; si on a une feuille, ce n'est pas notre problème ici, on la renvoie
    (case (racine A)
     ((+) (simplif+ A)) ; on passe la main à un simplificateur spécialisé dans les règles d'additions
     ((-) (simplif- A)); de même avec un simplificateur pour les règles de soustraction
     ((*) (simplif* A)) ; Il faut tout faire cas par cas. Ceci n'a rien à voir avec l'astuce de l'eval !
     ((/) (simplif/ A))
     (else (error "Operateur inconnu" (racine A))))))
```

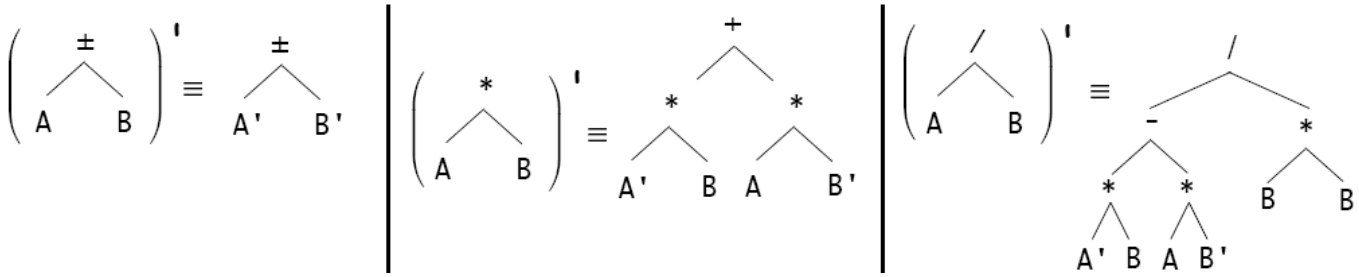
```
(define (simplif+ A)       ; on sait que A est un arbre de racine +
 (let ((A1 (simplif (fg A))) (A2 (simplif (fd A)))) ; hypothèse de récurrence, j'ai simplifié les sous-arbres
  (cond ((equal? 0 A1) A2) ; si je fais 0 + x, alors il ne me reste que x
        ((equal? 0 A2) A1)
        ((and (number? A1) (number? A2)) (+ A1 A2)) ; si ce sont deux nombres, j'applique la primitive +
        ((equal? A1 A2) (arbre '* 2 A1)) ; si je fais x + x alors je peux rassembler en 2*x. Discutable !
        (else (arbre '+ A1 A2))))) ; et sinon... je ne peux rien faire, je le laisse, sans moyens de simplifier

(define (simplif- A)       ; on sait que A est un arbre de racine -
 (let ((A1 (simplif (fg A))) (A2 (simplif (fd A))))
  (cond ((equal? 0 A2) A1) ; je soustrait A2 à A1. Donc si A2 est nul, cela revient à ne rien faire pour A1
        ((equal? A1 A2) 0) ; s'ils sont égaux, le résultat est 0
        ((and (number? A1) (number? A2)) (- A1 A2))
        (else (arbre '- A1 A2)))))
```

Notons que notre simplificateur n'est pas non plus optimal en raison de notre grammaire d'arbres strictement binaires. Par exemple, 0 – A1 ne sera pas simplifié car – A1 n'est pas considéré comme arbre.

V. La dérivation symbolique

Il s'agit ici de dériver un arbre d'expression par rapport à une variable. Comme précédemment, il y a des règles différentes à appliquer celui qu'on soit en présence d'un + ou -, etc. On orientera donc le dérivateur vers le dérivateur spécialisé. Les règles fondamentales sont les suivantes :




```
(define (derive A v)              ; retourne dA/dv en passant la main au dérivateur spécialisé
 (if (feuille? A)
    (if (equal? A v) 1 0)         ; dv/dv=1 et du/dv=0
    (simplif (case (racine A) ; la taille des arbres croît très vite en dérivant, il faut les simplifier !
               ((+ -) (derive+- A v))
               ((*) (derive* A v))
               ((/) (derive/ A v))
               (else (error "Op. inconnu" (racine A)))))))

(define (derive+- A v)            ; A = (+- Ag Ad)
 (arbre (racine A) (derive (fg A) v) (derive (fd A) v)))

(define (derive* A v)             ; A = (* Ag Ad)
 (let ((fgA (fg A)) (fdA (fd A)))
  (arbre '+ (arbre '* (derive fgA v) fdA) (arbre '* fgA (derive fdA v)))))

(define (derive/ A v)             ; A = (/ Ag Ad)
 (let ((fgA (fg A)) (fdA (fd A)))
  (arbre '/ (arbre '- (arbre '* (derive fgA v) fdA) (arbre '* fgA (derive fdA v))) (arbre '* fdA fdA))))
```

A partir du moment où on a calculé la dérivée première, on peut s'occuper des $n^{ièmes}$ par récurrence. Notons que $f^n = (f^{n-1})'$ est récursif tandis que $f^n = (f')^{n-1}$ est itératif : plus efficace, pas besoin de pile.

Comme on est capable de faire la dérivée n^{ième} d'un arbre et de prendre sa valeur en un point, on peut maintenant calculer la Série de Taylor. On rappelle que le développement de A(x) en x_0 à l'ordre n est :

$$\sum_{k=0}^{n} \frac{A^{(k)}(x_0)}{k!}(x - x_0)^k$$

```
(define (taylor A x x0 n) ; on renvoie la liste des coefficients de la série de Taylor
  (define (sub A n0) ; on calcule la somme de façon itérative. On part de 0 à n.
    (if (= n n0) ; si on est arrivé à n, alors on s'arrête. Sinon, on construit la liste des coefficients.
        '()
        (cons (/ (valeur A (list (list x x0))) (fac n0)) (sub (derive A x) (+ n0 1)))))
  (sub A 0))
```

VI. Une introduction à la compilation des arbres

A partir d'un arbre, on veut arriver à produire un code sur le principe de l'assembleur. Il faut donc définir le langage machine cible de la compilation. Commençons par un tour d'horizon des processeurs actuels :
- RISC : Reduced Instruction Set Computer. Il y a peu d'instructions mais elles sont très rapides. L'idéal pour des stations graphiques. Le compilateur produira plus d'instructions.
- CISC : Complex Instruction Set Computer. Beaucoup d'instructions de haut niveau. Le jeu d'instruction classique. Le compilateur produira un nombre raisonnable d'instructions.

La différence entre ces jeux d'instructions tend à se réduire : les approches convergent.

Nous allons traduire en langage machine nos arbres d'expressions avec une machine virtuelle à pile, dont le jeu d'instructions sera extrêmement réduit. Une fois le principe connu, ceux qui le veulent l'étendront !

Il faut commencer par définir une structure de donnée essentielle pour une compilation : la Pile.
On rappel que les piles fonctionnent en ordre LIFO : Last In, First Out. Nos piles seront fonctionnelles, dans le sens où une opération sur P ne la modifiera pas mais en calculera une nouvelle.
Pour avoir une bonne complexité, on utilise en interne une liste. Ainsi, tout sera en O(1) :

```
(define pile-vide '()) ; nos piles sont des listes. Une pile vide, c'est simplement la liste vide.

(define (pile-vide? P)
  (null? P))

(define (empiler x P) ; pour empiler, on fait un chaînage. C'est en O(1) dans ce sens.
  (cons x P))

(define (sommet P) ; on décale nos éléments. Le dernier élément inséré est donc le car
  (if (pile-vide? P)
      (error "le sommet de la pile est vide..")
      (car P)))

(define (depiler P)
  (if (pile-vide? P)
      (error "on ne peut pas depiler une pile vide !")
      (cdr P)))
```

Grâce à nos piles, on va pouvoir commencer par faire un parcours itératif d'arbre. On pouvait faire une itération avec un passage au CPS, mais on prenait sur le tas au lieu de la pile. La présence de deux fils introduisait des mises en attente : il suffira maintenant de les mettre dans une pile !

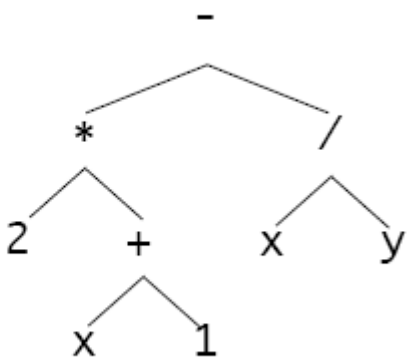
```
(define (nombre-feuilles A)
  (define (iter A P acc) ; arbre, pile et résultat courant
    (if (feuille? A)
        (if (pile-vide? P) ; si je suis sur une feuille, je regarde s'il me reste un arbre à visiter, i.e. sur la pile !
            (+ acc 1) ; je n'ai plus rien à visiter ? ok, j'ai fini ma part.
            (iter (sommet P) (depiler P) (+ acc 1))) ; sinon, je le visite en le sortant de la pile
        (iter (fg A) (empiler (fd A) P) acc))) ; je suis sur un nœud : je vais à gauche et j'empile à droite
  (iter A pile-vide 0))
```

Notons bien que lorsqu'on parle d'empiler un arbre, nous n'empilons qu'un pointeur sur l'arbre. Il n'y a en aucun cas recopie de toutes les données.

A présent, nous allons transformer un arbre d'expression vers un code pour notre machine virtuelle. L'arbre que l'on compile n'a que des opérateurs (-, *, …) et des opérandes (constantes ou variables). Le langage n'a donc besoin à ce stade que de deux instructions :

- (push x), on empile le sommet de x sur la pile
- (call op), on dépile le sommet pour avoir x1, puis on prend l'élément en dessous x2, et on empile le résultat de (op x2 x1). Par exemple : (push 5) (push 8) (call -) donne -3.

```
((push 2)
 (push x)
 (push 1)      compilation du fg
 (call +)
 (call *)
 ---------------------------------
 (push x)
 (push y)      compilation du fd
 (call /)
 ---------------------------------
 (call -))     call sur la racine
```




```
(define (compiler A)    ; Arbre --> Liste, version avec opérateurs binaires uniquement, pas de if !
  (if (feuille? A)
      (list (list 'push A))
      (append     (compiler (fg A))      ; notons que (append (list α) β) ≡ (cons α β)
                  (compiler (fd A))
                  (list (list 'call (racine A))))))
```

Un décompilateur est facile à produire à ce stade. Quand on voit un push, on fait tomber l'élément. Quand on a un call, on fait un arbre avec les deux derniers arbres obtenus. Exemple :

⌂ je vois (push 2). Je fais tomber 2.
⌂ je vois (push 1). Je fais tomber 1.
⌂ je vois (push x). Je fais tomber x.
⌂ je vois (call +). Je fais l'arbre de racine + et de fils x et 1.

```
(define (miniDecomp L P)      ; le décompilateur prend une liste et une pile vide
  (cond ((null? L) P)         ; si j'ai tout décompilé, je rend ma pile
        (else (case (caar L)  ; sinon je regarde quel est le mot clé auquel j'ai à faire
                ((push) (miniDecomp (cdr L) (empiler (cadar L) P))) ; c'est push : j'empile l'élément
                ((call) (let* ((x1 (sommet P))  (P1 (depiler P)) ; c'est call : je dépile les deux derniers éléments
                               (x2 (sommet P1)) (P2 (depiler P1)))
                          (miniDecomp (cdr L) (empiler (arbre (cadar L) x2 x1) P2)))) ; et j'en fais un arbre
                (else (error "Le décompilateur ne prend pas en charge ceci..."))))))) ; sinon… je ne connaît pas !
```

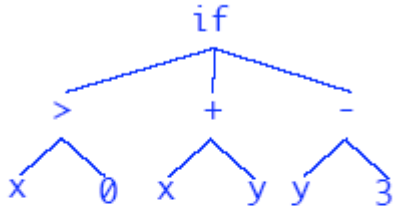
On s'intéresse maintenant à des structures un peu plus utiles, avec des nœuds conditionnels.
Un nœud if comporte trois fils : l'un est la condition, l'autre le résultat si le test donne faux, et le dernier le résultat si le test donne vrai. Notons qu'avec cet arbre à 3 fils, il faudra étendre la grammaire actuelle.

Le if réalise un aiguillage. Il faudra donc un mécanisme de branchement en langage machine. On utilise :

- des étiquettes (tags). Ils ne font rien, se contentant de signaler un endroit du code.
- Une instruction (brf etiq) qui réalise un branchement conditionnel. Si le sommet de pile est #f on se branche à l'étiquette, sinon on continue normalement l'exécution. Puisqu'on a interprété le sommet de la pile quoiqu'il en soit, on va dépiler.
- Une instruction (jmp etiq) de branchement inconditionnel. On saute à l'adresse, touche pas la pile.

(push x) (push 0) (call >)	*Compilation du fg*
(brf e1)	*Génération d'un BRF*
(push x) (push y) (call +)	*Compilation du fd*
(jmp e2)	*Génération d'un JMP*
e1	*Génération d'une étiquette*
(push y) (push 3) (call -)	*Compilation du fdd*
e2	*Génération d'une étiquette*

(if (> x 0) (+ x y) (- y 3))



- Attention : les `if` pouvant être emboîtés, chaque `if` devra posséder ses propres étiquettes !! On utilisera `(gensym 'etiq)`...

```
(define (compiler A)    ; arbres avec des nœuds if
 (cond ((feuille? A) (list (list 'push A)))
       ((equal? (racine A) 'if) (compiler-if A))    ; forme  speciale !
       (else (append (compiler (fg A))
                     (compiler (fd A))
                     (list (list 'call (racine A)))))))

; la compilation d'un noeud if commencera par celle du test
; pour aiguiller selon le sommet de pile sur l'arbre du centre ou  l'arbre tout à droite (fdd)
(define (compiler-if A)
 (let ((etiq1 (gensym 'etiq)) ; une première étiquette
       (etiq2 (gensym 'etiq))) ; ... et une seconde
   (append
    (compiler (fg A)) ; on compile le fils gauche.
    (list (list 'brf etiq1))
    (compiler (fd A)) ; on compile le fils droit...
    (list (list 'jmp etiq2))
    (list etiq1) ; append recolle des listes !
    (compiler (fdd A))
    (list etiq2))))
```

Il faudrait bien entendu optimiser le code produit, ce qui n'est pas fait ici. Par exemple, dans le code :

((push 2) (push 3) (jump e8) (push 45) e8 (push 7) ...)

On voit que le (push 45) ne sera jamais exécuté. Ceci est un exemple simple pour sensibiliser au problème de ce qu'on nomme les « branches mortes », des parties de code inutile. Ici, un remède simple est de supprimer tout ce qu'il y a entre un jump et la prochaine balise, car ce ne sera jamais accessible.
Il y aussi un jump sur une balise suivie elle-même d'un jump, qu'il faudrait relier directement…

```
(define (exec L P AL)                    ; code compilé x Pile x Mémoire --> Pile
 ;(printf "L=~a P=~a AL=~a\n" L P AL)      ; pour le debug...
 (cond ((null? L) P)
     ((symbol? (car L)) ....)            ; on passe les etiquettes...
     (else (case (caar L)
           ((push) (if (number? (cadar L))
                       (exec (cdr L) (empiler (cadar L) P) AL) ; si  c'est un nombre, direct
                       (let ((valeur (giveValeur (cadar L)  AL))) ;sinon, valeur de la variable
                         (exec (cdr L) (empiler valeur P) AL)))) ; et  j'empile cette valeur
           ((call) (let* ((x1 (sommet P))
                          (P1 (depiler P))
                          (x2 (sommet P1))
                          (P2 (depiler P1)))
                     (exec (cdr L) ; j'ai la fonction à appliquer  sur les deux données
                           ;(printf "x2 = ~a ; x1 = ~a ; opérateur  = ~a\n" x2 x1 (cadar L))
                           (empiler ((eval (cadar L)) x2 x1) P2) ;  bourrin... ne fonctionne qu'au top-level
                           AL)))
           ((jmp) (exec (cdr (member (cadar L) (cdr L))) P AL))  ; on cherche où se placer
           ((brf) (if (sommet P)
                      (exec (cdr L) (depiler P) AL)  ; si c'est faut on continue
                      (exec (cdr (member (cadar L) (cdr L)))  (depiler P) AL))) ; sinon on cherche où se placer
           (else (error "Instruction inconnue" (car L)))))))
```

Chacun aura sa façon de faire le décompilateur. On propose ici celle de Vincent Ardisson :

```
(define (without-last-cdr L)
  (define (iter L res)
    (if (or (null? L) (null? (cdr L)) )
        (reverse res)
        (iter (cdr L) (cons (car L) res))))
  (iter L '()))

(define (decompiler programme) ; point d'entrée
  (define (look-for x L)
    (define (iter L res)
      (cond ((null? L) (reverse res))
            ((equal? x (car L)) (reverse res))
            (else (iter (cdr L) (cons (car L) res)))))
    (iter L '()))

  (define (iter A1 programme)
    (printf "~a ~a\n" A1 programme)
    (cond ((null? programme) (sommet A1))
          ((symbol? (car programme)) (iter A1 (cdr programme)))
          (else (case (caar programme)
                  ((push) (iter (empiler (cadar programme) A1) (cdr programme)))
                  ((call) (let* ((s1 (sommet A1)) (p1 (depiler A1)) (s2 (sommet p1)) (p2 (depiler p1)))
                            (if (and (not (null? (cdr programme))) (equal? (caadr programme) 'brf))
                                (let* ((etiq1 (cadadr programme))
                                       (tmp (look-for etiq1 programme))
                                       (etiq2 (cadar (last-pair tmp)))
                                       (codet (cddr (without-last-cdr tmp)))
                                       (codef (cdr (member etiq1 (look-for etiq2 programme))))
                                       (restecode (cdr (member etiq2 programme))))
                                  (printf "codet ~a - ~a\n" codef codet)
                                  (iter (empiler (arbre 'if (arbre (cadar programme) s2 s1) (decomp1 codet)
                                                 (decomp1 codef)) p2) restecode))
                                (iter (empiler (arbre (cadar programme) s2 s1) p2) (cdr programme)))))
                  (else (iter A1 (cdr programme)))))))
  (iter '() programme))
```

VII. La curryfication

La fonction (valeur A AL) pour calculer la valeur de l'arbre A avec la mémoire AL est (trop) simple :

- compilation de l'arbre A et production d'un programme L dans notre langage
- exécution du programme L sur une pile
- récupération du résultat au sommet de la pile

```
(define (valeur A AL)
  (sommet (exec (compiler A) pile-vide AL)))
```

Haskell Curry (1900-1982)
Mathématicien états-unien. Concept de curryfication, correspondance de Curry-Howard. Grand en logique combinatoire.

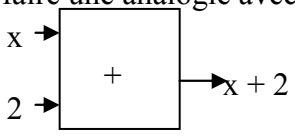
Si on souhaite calculer la valeur de l'arbre A dans plusieurs contextes de mémoire, il y a un problème :
(valeur '(- (* 2 (+ x 1)) (/ x y)) '((x 4) (y 2)))
(valeur '(- (* 2 (+ x 1)) (/ x y)) '((x 0) (y 2)))
On a exactement le même programme pour des données différentes, mais on le compile deux fois !
La solution consiste à faire une curryfication.

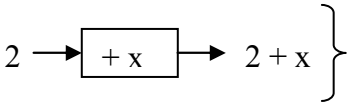
La curryfication désigne l'opération qui fait passer d'une fonction à plusieurs arguments à une fonction à un argument, et qui retourne une fonction prenant le reste des arguments ; l'opération inverse est évidemment possible. Par exemple, f(x, y) → x+2y peut-être considéré comme une fonction à une seule variable, qui est dans $|R^2$: on donne le couple (3, 2) considéré comme une seule entité.

```
(define (curry f)  ; curryfication d'une fonction f binaire
  (lambda (x)
    (lambda (y)
      (f x y))))
```

```
(define add (curry +))
(add 4)        → #<procedure>
((add 4) 5)    → 9
```

On peut faire une analogie avec les circuits :





Le compilateur CAML génère le circuit + x, l'additionneur de x. Sa machine virtuelle profite qu'il n'y a que des fonctions à 1 variable (1 argument) en CAML

VIII. Quelques développements

Les programmes suivant sont issues de l'imagination fertile de J.-P. Roy :

- une fonction disant si une séquence donnée est bien un programme tel que nous l'avons défini (#f, #t) ; c'est un checker de bytecode.
- De manière itérative, fabriquer un arbre aléatoire de hauteur donnée
- Etant donné un arbre, tirer un sous arbre au hasard sans calculer les feuilles
- Pour avoir des sauts en temps constant, remplacer les étiquettes par des numéros de case (tableau)

Notons que la compilation est bijective. On a un algorithme pour compiler, et comme la décompilation est l'inverse exacte de la compilation alors on devrait pouvoir en déduire automatiquement l'algorithme de décompilation. Ceci est un travail de recherche chez les russes ou les thésards de Kounalis…

i

[1] Ceci sont différentes façons de lire un arbre, mais il s'agit de variantes d'un parcours en profondeur. Il existe aussi des parcours en largeur, qui sert par exemple dans le cas d'une contrainte demandant de trouver quelque chose le plus proche possible de la racine ; c'est par exemple utile dans un graphe, couplé avec des contraintes de type Δ-D.

[2] Notons que la mémoire peut contenir non seulement des associations entre des variables et des valeurs, mais aussi entre des fonctions et des procédures. On peut stocker dans la A-liste des fonctions personnelles déclarées en λ, et aller les chercher si la fonction que l'on cherche à appliquer n'est pas connue. Attention, cette stratégie empêche le recours à un eval ; case obligé.

[3] Un texte fondateur est « Algebraic Simplification, a guide for the perplexed » de Joel Moses, Project MAC, MIT 1971. Notons que la simplification d'un arbre doit être de surcroît dirigé par ce que l'on compte faire après. Par exemple, si on a f + g, faut-il laisser ou rassembler ? Si on fait dériver, il faudrait mieux laisser sous cette forme, sachant qu'il est plus facile de dériver des sommes…