

ENSEIRB-MATMECA

SEMESTRE 6

PROJET SCHEME

Projet Code Generation

I1

Alexandre HERVE

Maëva GRONDIN

Nicolas HANNOYER

Simon BROUARD



Mardi 10 Mai 2016

SOMMAIRE

1	Chaîne d'additions - algorithme d'exponentiation	2
1.1	Définition d'une chaîne d'additions	2
1.2	Vérification chaîne d'additions	2
1.3	L'algorithme d'exponentiation	3
1.3.1	Stockage des résultats intermédiaires	3
1.3.2	La fonction exponentiation	4
2	La génération de chaîne d'addition	5
2.1	La stratégie binaire	5
2.2	La stratégie décimale	6
2.3	La méthode euclidienne	6
2.4	La décomposition en facteurs premiers	6
3	La production d'algorithmes	7
3.1	Principe de la production d'algorithmes	8
3.1.1	Stockage des calculs intermédiaires	8
3.2	Algorithmes en schéma	9
3.2.1	Code à générer	9
3.2.2	Génération du corps du let*	9
3.2.3	Résultats	10
3.3	Algorithmes en C	10
3.3.1	Code à générer	10
3.3.2	Stockage des variables	11
3.3.3	Génération d'une affectation	12
3.3.4	Génération du code c complet	12
3.3.5	Résultats	13
3.3.6	Améliorations	14
3.4	Calcul du nombre de registres nécessaires au calcul	14
4	Conclusion	15

1 Chaîne d'additions - algorithme d'exponentiation

1.1 Définition d'une chaîne d'additions

Une chaîne d'additions pour un entier n est une liste

- qui ne contient que des entiers, organisées de manière strictement croissante
- qui commence par 1
- dont chaque élément (hormis 1) est la somme de deux éléments précédents de la liste
- dont le dernier élément est n

exemple : '(1 2 3 5 7 10) est une chaîne d'additions pour l'entier 10.

L'intérêt d'une telle liste est de pouvoir facilement pour un entier x calculer x^n . Ce projet se base sur cet objet. Notre objectif est dans un premier temps de générer des chaînes d'additions pour un entier n , puis de générer automatiquement des fonctions d'exponentiation en fonction d'un entier n donné en langage C et scheme. La génération de fonction d'exponentiation utilisera évidemment les chaînes d'additions générées.

1.2 Vérification chaîne d'additions

Pour réaliser un calcul d'exposant ou pour générer une fonction d'exponentiation à partir d'une liste l , il peut être intéressant de vérifier que cette liste est bien une chaîne d'additions. En effet, les algorithmes se basant sur les chaînes d'addition, si l n'en est pas une cela pourrait faire dysfonctionner le programme.

La fonction is-addlist : Pour vérifier que l est une chaîne d'additions il suffit de vérifier qu'elle a en toutes les caractéristiques, on vérifie donc dans l'ordre que:

- l n'est pas vide : (not? (null? l))
- l commence bien par 1 : (= 1 (car l))
- les éléments de l sont bien entiers et rangés dans l'ordre croissant : (is-growing? (list (car l)) (cdr l))
- chaque élément de l est bien la somme de deux éléments précédents de l : (all-are-sum? l)

La fonction is-addlist prend en argument une liste d'entiers l . Si l'une de ces conditions n'est pas vérifiée elle retourne #f, sinon elle retourne #t.

La fonction is-growing prend en argument une liste d'entiers et vérifie simplement que chaque élément est soit le dernier de la liste, soit plus petit que son successeur. Elle retourne #t si c'est le cas, #f sinon.

La fonction all-are-sum? prend en argument une liste qui prend en argument deux listes d'entiers l1 et l2. Et retourne #t si tous les éléments de l2 sont la somme de deux éléments de l1 ou le précédent dans l2. Son principe est le suivant :

- Si l2 est vide : #t
- Sinon :
 - Si (car l2) n'est pas la somme de deux éléments de l1 (possiblement les mêmes) (is-sum? l1 (car l1)): #f
 - Sinon on rappelle is-growing? avec en premier argument l1 à laquelle on a rajouter le premier élément à la fin, et en deuxième argument l2 privée de son premier élément.

La fonction is-sum? prend en argument une liste d'entier l et un entier x, et retourne #t si x un la somme de deux éléments de l, #f sinon. Elle est appelée sur tout les éléments de la potentielle chaîne d'additions (sauf 1) par all-are-sum?

1.3 L'algorithme d'exponentiation

Il s'agit à présent d'écrire l'algorithme permettant de calculer x^n à partir de la chaîne d'additions pour l'entier n, et de l'entier x.

Exemple : à partir de la chaîne d'additions '(1 2 4) on peut calculer x^4 de la manière suivante: $x^4 = x^2 * x^2$ avec $x^2 = x * x$, donc $x^4 = (x * x) * (x * x)$

On remarque ici que pour calculer x^4 on calcule deux fois x^2 . Nous souhaiterions éviter de refaire plusieurs fois les mêmes calculs pour optimiser l'algorithme d'exponentiation. Pour cela nous devons stocker tous les résultats intermédiaires.

1.3.1 Stockage des résultats intermédiaires

Nous allons transformer la chaîne d'additions en alist avec la fonction **create-list-of-list**.

Exemple : '(1 2 3 5 8 13) \longrightarrow '((1) (2) (3) (5) (8) (13))

Cette fonction prend en argument deux listes d'entiers (initialement la même chaîne d'additions). Puis à chaque fois que l'on calcule un résultat intermédiaire x^n on le stocke à la fin de la sous liste correspondante (qui commence par n) grâce à la fonction **stock**.

Exemple : si on cherche à calculer 2^{13} et que l'on a déjà calculé $2^1 = 2$, $2^2 = 4$ et $2^3 = 8$ on a la liste : '((1 2) (2 4) (3 8) (5) (8) (13))

La fonction stock prend en argument un entier res qui est le résultat à stocker, un entier elt qui est l'élément de la chaîne d'additions derrière lequel on va stocker res et une alist l.

Exemple : (stock 4 2 '((1) (2) (3))) \longrightarrow '((1) (2 4) (3))

Le principe de cette fonction est tout simplement de remplacer dans l la liste contenant elt par la liste '(elt res). Dans l'exemple ci-dessus on remplace '(2) par '(2 4).

1.3.2 La fonction exponentiation

La fonction exponentiation-init prend en argument un entier x dont on veut calculer une puissance, et une liste l qui est la chaîne d'additions à partir de laquelle on va faire ce calcul. Elle va créer une aliste $l2$ à partir de l grâce à la fonction `create-list-of-list`, et retourner `(exponentiation-stock x l l2)`.

La fonction exponentiation-stock prend en argument un entier x , une liste l qui est une chaîne d'addition, et une aliste `lcomplete` qui est issue de l . Elle retourne x^n . Le principe est le suivant :

On vérifie d'abord que l est une chaîne d'additions, si c'est bien le cas alors :

- Si l ne contient qu'un élément (1) : retourner x
- Sinon $p \leftarrow$ paire d'entiers de l dont la somme vaut `(last l)` // (cf. fonction `sum-of`)
 - Si le premier élément de $p = 0$: $a \leftarrow 1$;
 - Sinon si le premier élément de $p = 1$: $a \leftarrow x$;
 - Sinon a prend la valeur du retour de `exponentiation-stock` appelé sur :
 - * x
 - * la liste l dont on aura enlevé tous les éléments plus grand que le premier élément de p (cf. `delete-greater`)
 - * `lcomplete`
 Le retour de cette fonction étant au passage stocké dans `lcomplete` grâce à la fonction `stock`
 - De même pour le deuxième élément de p , on notera b le résultat.
 - retourner $a * b$;

Explication de l'algorithme : En supprimant un élément de l à chaque appel de la fonction on s'assure qu'elle termine. En effet, elle se termine si l ne contient plus qu'un élément, obligatoirement l'élément 1 car c'est une chaîne d'addition. Et comme $x^1 = x$ elle retourne alors x . Pour comprendre le reste de l'algorithme, mieux vaut regarder un exemple :

- $(\text{exponentiation-stock } 2 '(1\ 2\ 3\ 5\ 7) '((1)\ (2)\ (3)\ (5))) = (\text{exponentiation-stock } 2 '(1\ 2) '((1)\ (2)\ (3)\ (5))) * (\text{exponentiation-stock } 2 '(1\ 2\ 3) '((1)\ (2)\ (3)\ (5)))$
- $(\text{exponentiation-stock } 2 '(1\ 2) '((1)\ (2)\ (3)\ (5))) = (\text{exponentiation-stock } 2 '(1) '((1)\ (2)\ (3)\ (5))) * (\text{exponentiation-stock } 2 '(1) '((1)\ (2)\ (3)\ (5)))$
 - * $(\text{exponentiation-stock } 2 '(1) '((1)\ (2)\ (3)\ (5))) = 2$; `lcomplete` devient `'((1\ 2)\ (2)\ (3)\ (5))`
 - * $(\text{exponentiation-stock } 2 '(1) '((1\ 2)\ (2)\ (3)\ (5))) = 2$;
 - $2 * 2 = 4$; `lcomplete` devient `'((1\ 2)\ (2\ 4)\ (3)\ (5))`
 - $(\text{exponentiation-stock } 2 '(1\ 2\ 3) '((1\ 2)\ (2\ 4)\ (3)\ (5))) = (\text{exponentiation-stock } 2 '(1)\ ((1\ 2)\ (2)\ (3)\ (5))) * (\text{exponentiation-stock } 2 '(1\ 2)\ ((1\ 2)\ (2\ 4)\ (3)\ (5)))$
 - * $(\text{exponentiation-stock } 2 '(1) '((1\ 2)\ (2\ 4)\ (3)\ (5))) = 2$
 - * $(\text{exponentiation-stock } 2 '(1\ 2)\ ((1\ 2)\ (2\ 4)\ (3)\ (5))) = 4$

$2 * 4 = 8$; lcomplete devient '(1 2) (2 4) (3 8) (5))

$4 * 8 = 32$; lcomplete devient '((1 2) (2 4) (3 8) (5 32))

On obtient bien $2^5 = 32$.

La fonction sum-of prend en argument un entier x et une liste d'entiers l (chaîne d'additions) et retourne $\#t$ si x est la somme de deux éléments de l , $\#f$ sinon.
sum-of compare x et la somme du premier et dernier élément de l .

- Si $x = (\text{premier élément de } l) + (\text{dernier élément de } l)$, retourner la liste (premier élément de l , dernier élément de l)
- Sinon si $x < (\text{premier élément de } l) + (\text{dernier élément de } l)$, on rappelle sum-of en supprimant le dernier élément de l .
- Si $x > (\text{premier élément de } l) + (\text{dernier élément de } l)$, on rappelle sum-of en supprimant le premier élément de l .

Exemple : (sum-of 7 '(1 2 3 4 5)) \longrightarrow '(2 5)

La fonction delete-greater prend en argument une liste l un entier x (qui doit être dans l). Et retourne la liste l dont elle aura supprimé tous les éléments strictement supérieurs à l .

Exponentiation naïf : Nous avons dans un premier temps écrit une fonction d'exponentiation qui utilisait le même principe de calcul, mais qui ne stockait pas les résultats intermédiaires et refaisait donc plusieurs fois le même calcul. Ce qui ne la rendait pas assez performante, c'est pourquoi nous avons décidé de créer une nouvelle version de cette fonction.

2 La génération de chaîne d'addition

Le but de cette partie est de générer la meilleure chaîne d'additions c'est à dire la chaîne d'additions de longueur minimale pour un entier $n \geq 1$ donné. Pour se faire, plusieurs approches sont possibles, nous verrons ici quatre méthodes de génération.

2.1 La stratégie binaire

La stratégie binaire consiste à diviser par 2 les nombres pairs et soustraire 1 aux nombres impairs. Prenons l'exemple de la chaîne d'additions de 142 :

142 est pair $\rightarrow 71$;
71 est impair $\rightarrow 70$;
70 est pair $\rightarrow 35$;
35 est impair $\rightarrow 34$;
34 est pair $\rightarrow 17$;
17 est impair $\rightarrow 16$;
16 est pair $\rightarrow 8$;
8 est pair $\rightarrow 4$;
4 est pair $\rightarrow 2$;
2 est pair $\rightarrow 1$;

On obtient finalement la chaîne d'additions '(1 2 4 8 16 17 34 35 70 71 142) de longueur 11. Cette algorithme est de complexité logarithmique en temps du fait de la division par 2. En effet, dans le pire des cas, pour chaque division on fait une soustraction, on multiplie donc le nombre d'opérations par 2 ce qui ne change pas la complexité. Sa longueur est ainsi inférieure à $2 \times \log_2(n)$.

2.2 La stratégie décimale

La stratégie décimale consiste à décomposer le nombre en somme de puissance de 10 et de générer chaque puissance de 10 le plus rapidement possible.

Reprenons l'exemple de 142:

$2 \rightarrow '(1\ 2)$

$4 \rightarrow '(1\ 2\ 4)$

$1 \rightarrow '(1)$

Maintenant on doit les assembler. Pour cela il faut compléter les chaînes de puissance de 10 inférieures.

$'(1\ 2) \rightarrow '(1\ 2\ 4\ 8)$

$'(1\ 2\ 4) \rightarrow '(1\ 2\ 4\ 8)$

Ensuite on multiplie chaque terme des chaînes intermédiaires par la puissance de 10 qui l'accompagne.

$'(1\ 2\ 4\ 8) \rightarrow '(1\ 2\ 4\ 8)$

$'(1\ 2\ 4\ 8) \rightarrow '(10\ 20\ 40\ 80)$

$'(1) \rightarrow '(100)$

On génère ensuite la chaîne qui additionne les différentes puissances de 10: '(140 142)

On n'a plus qu'à concaténer les chaînes, ce qui donne '(1 2 4 8 10 20 40 80 100 140 142)

Pour accéder à la puissance supérieur on doit générer une chaîne de longueur 4. On obtient donc une chaîne beaucoup plus grande que celle de la stratégie binaire si le nombre donné en entrée est grand.

2.3 La méthode euclidienne

Le principe de la méthode euclidienne est de réutiliser l'algorithme de la division euclidienne. On part de 2 entiers et on soustrait les deux pour obtenir un reste qui sera réutilisé dans l'appel récursif. Cette méthode génère une chaîne dont la longueur dépend du deuxième entier donné en entrée. On génère donc pour tout entier k inférieur à n les chaînes obtenue à partir des entiers n et k . On obtient donc une liste de chaîne et on retourne la plus petite en sortie de fonction.

Cette méthode a l'avantage de générer des chaînes proches de la chaîne minimale. Mais cette algorithme a un grand coût en temps et en espace.

2.4 La décomposition en facteurs premiers

Dans cette méthode, on détermine d'abord les facteurs premiers de n et leur exposant dans la décomposition en facteurs premiers. On calcul la chaîne d'additions de chacun des facteurs premiers et on en génère autant que leur exposant. Une fois que toutes les chaînes sont calculées, on appelle récursivement les chaînes d'additions par ordre décroissant de facteurs. Sauf pour la première chaîne, on retire le 1 des chaînes. À chaque appel on multiplie la chaîne à ajouter par le dernier terme de la chaîne principale. On concatène alors les deux chaînes.

Pour éclaircir cette méthode calculons la chaîne d'additions de 180:

Les facteurs premiers de 180 sont 2, 3 et 5 d'exposant respectif 2, 2 et 1.

Les chaînes d'additions:

$2 \rightarrow '(1\ 2)$
 $3 \rightarrow '(1\ 2\ 3)$
 $5 \rightarrow '(1\ 2\ 4\ 5)$

d'où

$'(1\ 2\ 3\ 4\ 5)$
 $\quad '(1\ 2\ 3)$
 $\quad\quad '(1\ 2\ 3)$
 $\quad\quad\quad '(1\ 2)$
 $\quad\quad\quad\quad '(1\ 2)$

Et on obtient $'(1\ 2\ 4\ 5\ 2*5\ 3*5\ 2*(3*5)\ 3*(3*5)\ 2*(3*3*5)\ 2*(2*3*3*5))$.

Ce qui donne $'(1\ 2\ 4\ 5\ 10\ 15\ 30\ 45\ 90\ 180)$.

Cette méthode n'a pas pu être implémentée car la fonction qui permet de récupérer la liste des facteurs premiers et la liste des exposants n'est pas présente dans le langage racket, de plus nous n'avons pas créé l'équivalent de cette fonction en racket. Par conséquent la comparaison avec les autres méthodes n'a pas été faite. En revanche la complexité en temps sera forcément plus lourde que la stratégie binaire en raison du calcul des facteurs premiers.

3 La production d'algorithmes

Le but de cette partie est de générer automatiquement des fonctions d'exponentiation (en langage C et en langage scheme). C'est à dire écrire un algorithme qui à partir d'un entier n créer une fonction exp_n qui prend en argument un entier x et retourne x^n .

$$generer_fonction(n) \longrightarrow exp_n(x) \longrightarrow x^n$$

La fonction générée prend également un opérateur *mult* en argument (* dans le cas de la fonction d'exponentiation), permettant ainsi d'utiliser la fonction exp_n générée pour d'autres opérations. Par exemple :

$$\begin{aligned}
 generer_fonction(op, n) &\longrightarrow exp_n(op, x) \longrightarrow op(x, n) \\
 \text{fonction d'exponentiation : } &exp_n(*, x) \longrightarrow x^n \\
 \text{fonction multiplication : } &exp_n(+, x) \longrightarrow x * n
 \end{aligned}$$

L'un des objectifs de cette partie est également que la fonction générée soit totalement optimisée, ce qui en fait son intérêt, c'est à dire :

- Qu'elle ne fasse que des opérations nécessaires au calcul de x^n ($op(x, n)$ dans le cas général. Pour cela nous utilisons les chaînes d'additions générées à partir de n et des algorithmes créés précédemment.
- Qu'elle ne refasse aucune opération déjà effectuée lors du même appel à cette fonction, c'est ce qui fait toute la difficulté cette génération de fonctions. Il faudra donc que la fonction obtenue stocke les résultats des calculs qu'elle a effectuée et dont elle aura encore besoin.

Les principes de construction des fonctions en scheme et en C sont similaires. Mais le C étant un langage impératif et le scheme un langage fonctionnel, avec des syntaxes différentes, leur mise en œuvre diffère, notamment dans l'écriture finale de la fonction.

3.1 Principe de la production d'algorithmes

La fonction qui génère la fonction d'exponentiation ne prend en argument qu'un entier n . La première étape est donc de générer une chaîne d'additions à partir de n pour connaître précisément quels résultats intermédiaire la fonction exp_n devra calculer pour arriver à x^n sans faire de calculs superflus.

exemple : $generation_addlist(6) \rightarrow '(1\ 2\ 3\ 6)$

Dans l'exemple ci-dessus on générera la fonction exp_6 qui pour calculer x^6 réalisera les opérations suivantes :

$$\begin{aligned}x2 &\leftarrow x1 * x1 \\x3 &\leftarrow x2 * x1 \\x6 &\leftarrow x3 * x3\end{aligned}$$

On voit ici que chaque variable est utile pour aboutir au résultat final.

3.1.1 Stockage des calculs intermédiaires

Le premier problème que l'on rencontre est que la chaîne d'additions que l'on a obtenue ne nous donne pas la suite d'opérations à effectuer pour aboutir au résultat final, mais seulement la liste des résultats intermédiaires que l'on doit obtenir pour cela. Il faut donc dans un premier temps déterminer quelles sont ces opérations intermédiaires, et les stocker pour pouvoir les récupérer lors de l'écriture de la fonction en fonction du langage. Ceci est fait par les fonction *addlist2list-of-list* et *create-sumlist*. Elles transforment une chaîne d'additions en alist dont chaque sous liste est composée de trois élément. Le premier est l'élément qui se trouvait dans la chaîne d'additions (c'est à dire le résultat intermédiaire à calculer). Et le deuxième et troisième sont les deux éléments de la chaîne d'additions qu'il faut sommer pour aboutir au premier. (Exception faite de la première sous liste où 1 devient '(1 1 0), en effet aucune somme de deux éléments de la chaîne d'additions ne donne 1, mais comme toute chaîne d'additions commence par 1 on peut se permettre de fixer ce résultat : $1 = 1 + 0$)

exemple : $addlist2list-of-list('(1\ 2\ 3\ 6)) \rightarrow '((1\ 1\ 0)\ (2\ 1\ 1)\ (3\ 2\ 1)\ (6\ 3\ 3))$

Le principe le suivant : *addlist2list-of-list* retourne la concaténation du retour de *create-sumlist* appelé sur chaque élément de la chaîne d'additions.

create-sumlist est une fonction récursive qui prend en argument :

un entier n et une liste l (lors du premier appel l est la chaîne d'additions restreinte aux éléments inférieurs à).

- Si la somme du premier et du dernier élément de l donne n , retourner la liste formée de n et de ces deux éléments;
- Sinon si cette somme est supérieure à n alors retourner *create-sumlist*(n, l privée de son dernier élément)
- Sinon si cette somme est inférieure à n alors retourner *create-sumlist*(n, l privée de son premier élément)

exemple : $create-sumlist(5, '(1\ 2\ 4)) \rightarrow '(5\ 1\ 4)$

Cette fonction ne termine que si n est issue d'une chaîne d'additions et le une liste correspondant à tous les éléments inférieurs à n dans cette chaîne d'additions. On n'appelle pas cette fonction pour $n = 1$ car comme on l'a vu c'est un cas particulier. Une fois cette nouvelle liste créée, nous pouvons commencer la génération de code à proprement parler.

3.2 Algorithmes en scheme

3.2.1 Code à générer

Dans le cas du scheme nous cherchons à générer du code de type :

```
(define (exp3 mult x)
  (let* ([x2 (mult x x)]
        [x3 (mult x2 x)]
        x3)))
```

Ceci est fait sous la forme d'une liste. La fonction qui ce code est alors séparée en deux parties :

- Une fonction qui retourne l'entête de la fonction et le `let*`. Nous créons deux fonctions pour écrire cet entête de deux manières :
 Une lambda fonction, ex : `(lambda (op x) (let*))`
 Une fonction à laquelle on donne un nom pour pouvoir l'utiliser plusieurs fois plus facilement, ex : `(define (exp3 mult x) (let*))`
- Une fonction qui retourne le corps du `let*`. Ex : `'([x2 (mult x x)] [x3 (mult x2 x)] x3)`

La concaténation de ces deux listes donne la fonction à générer. La liste contenant l'entête de la fonction est obtenue par de simples concaténations et conversions de nombres, caractères, symboles et listes. Nous nous intéresserons donc plus particulièrement à la génération du corps du `let*`.

3.2.2 Génération du corps du `let*`

Pour générer le corps du `let*` on utilise la fonction *constr – let* qui prend en argument la alist l construite précédemment et un opérande op . Cette fonction est récursive et construit une nouvelle liste selon le principe suivant : notons $'(a\ b\ c)$ le car de l à chaque appel

- Si l est vide, retourner `'()`
- Si $(= 1\ a)$, retourner `(cons(listx1x)constr – let(op, (cdrl))`
- Sinon, retourner `(cons (list xa (list op xb xc)))`

x_b et x_c ayant nécessairement été créées avant x_a dans le corps du `let*` retourné car b et c arrivent avant a dans la chaîne d'additions.

Exemples :

```
(constr-let '* '(5 2 3)) → '((x5 (* 2 3)))
```

```
(constr-let '* '((1 0 0) (2 1 1) (4 2 2)) → '((x1 x) (x2 (* x1 x1)) (x4 (* x2 x2)))
```

Il ne reste alors plus qu'à rajouter à la fin de cette liste le retour du let* (respectivement x5 et x4 dans les deux exemples ci-dessus). Et en construisant la liste finale représentant la fonction générée à partir des listes que l'on vient de créer. Cela est respectivement avec la fonction *finale* – *funct* pour la fonction avec un nom, et la fonction *gen2* – *final* pour la lambda fonction.

3.2.3 Résultats

Il suffit alors, pour différents n , de lire le code de la fonction obtenue, et l'appliquer à différents paramètres pour constater que l'on obtient bien le résultat attendu.

```
(finale-funct 3 'op)
(define (exp3 op x) (let* ((x1 x) (x2 (op x1 x1)) (x3 (op x1 x2))) x3))
(exp3 * 2) ; 2³=8

(gen2-final 'op 5)
((lambda (mult x) (let* ((x1 x) (x2 (mult x1 x1)) (x3 (mult x1 x2)) (x5
```

Language: racket; memory limit: 128 MB.

```
'(define (exp3 op x) (let* ((x1 x) (x2 (op x1 x1)) (x3 (op x1 x2))) x3))
8
'(lambda (op x) (let* ((x1 x) (x2 (op x1 x1)) (x3 (op x1 x2)) (x5 (op x2
32
> |
```

Fig. 1: exemple de fonctions générées et de leur application

3.3 Algorithmes en C

Pour la génération de code en C nous utilisons des `fprintf` pour écrire directement dans un fichier `.c`

Une contrainte nous est ici rajouté par rapport à la génération de code scheme : la fonction générée doit avoir une complexité en espace minimale. Et en distinguant deux cas :

- Une fonction qui fait des multiplications in-place
- Une fonction qui ne fait que des multiplications out-place

Il faudra donc savoir quelle sont les variables qui peuvent être réutilisées, pour en utiliser exactement le minimum qu'il n'en faut pour le fonctionnement de l'algorithme.

3.3.1 Code à générer

La figure 2 montre un exemple de code C que l'on cherche à générer, ici une fonction avec une multiplication in-place. On remarque notamment la réutilisation de la variable `x1` pour stocker x^3 .

```

#include <stdio.h>
#include <stdlib.h>

int exp3(int x){
    int x1 = x * 1;
    int x2 = x1 * x1;
    x1 *= x2;
    return x1;
}

```

Fig. 2: exemple de code c que l'on souhaite générer

3.3.2 Stockage des variables

Pour ne pas utiliser plus de variables que nécessaire, il faut avant de générer chaque affectation savoir si l'on va créer une nouvelle variable, ou si une variable créée précédemment n'est plus utilisée et est donc disponible. De plus il faut conserver en mémoire pour chaque n de la chaîne d'additions quelle variable correspond à x^n , en effet ce n'est plus forcément x_n contraire aux variables utilisées dans la fonction *scheme*.

Stockage des variables dans la liste

Pour savoir quelle variable correspond à chaque élément de la chaîne d'additions on va modifier à chaque affectation la *alist* créée par *addlist2list - of - list*.

Par exemple si on a la *alist* '((2 1 1) (3 2 1) (4 2 2)), après l'affectation $x_2 \leftarrow x^2$ la *alist* devient '((2 1 1) (3 x2 1) (4 x2 x2)). Ceci est réalisé par la fonction *modify - variable - name* qui prend en argument :

- La *alist* dans laquelle on veut faire la modification.
- n (ici un élément de la chaîne d'additions, donc un entier, mais ce peut également être un symbole).
- Le symbole de la nouvelle variable.

Ainsi pour chaque élément '(a b c) de la *alist*, lorsque l'on arrive à l'affectation correspondant b et c ont déjà été remplacées par le nom de la variable correspondant. Il ne reste alors plus qu'à trouver une variable pour y stocker x^a et à écrire dans le fichier .c l'affectation

$$\text{nom_de_la_variable_}x^a = b * c;$$

Stockage des variables disponibles

Nous allons stocker dans une liste *variable - list* toutes les variables qui ont été déclarées au préalable et qui ne sont plus utilisées (la valeur qu'elle contient n'apparaît pas dans le reste de la *alist*). Pour cela on utilise la fonction *is - reused - variable?* qui prend en argument n un nom de variable et l la fin de la *alist* (après l'affectation en cours de traitement). Et qui retourne *#t* si la variable n est réutilisée et *#f* si elle est disponible pour stocker un autre résultat.

Si une variable n'est plus utilisée est sera soit immédiatement utilisée pour faire une multiplication in-place, soit stockée dans *variable - list*

3.3.3 Génération d'une affectation

Pour chaque élément de la *alist* on gère le stockage des variable ainsi que l'écriture de l'affectation correspondante dans le fichier *.c*. Notons $(a\ b\ c)$ un élément de la *alist*. Il faut distinguer plusieurs cas :

- Si *variable – list* n'est pas vide alors on stocke *a* dans (car *variable – list*) et on retire cette variable de *variable – list* qui devient $(\text{cdr } \textit{variable – list})$ (ou $'()$).
Si *b* et/ou *c* n'est plus utilisée ou l'ajoute à *variable – list*.
- Si *variable – list* est vide est vide et *b* et *c* sont réutilisées, on déclare une nouvelle variable "xa" et on effectue l'affectation $\text{int } xa = b * c;$
- Sinon (au moins un des variables *b* et *c* n'est plus utilisée et *variable – list* est vide), il faut distinguer le cas où les multiplications sont out-place du cas où l'on autorise les multiplications in-place.

Multiplication out-place

- On déclare une nouvelle variable "xa" et on effectue l'affectation $\text{int } xa = b * c;$
Si les variables *b* ou (/et) *c* ne sont plus utilisées on la (/les) stocke dans *variable – list*.

Multiplication in-place

- Si *b* n'est plus utilisée, on effectue l'affectation $b * = c;$
Si *c* n'est plus utilisée on l'ajoute à *variable – list*.
- Sinon (*b* et réutilisée mais pas *c*), on effectue l'affectation $c * = b;$

Dans chaque cas on modifie la *alist* en remplaçant *a* par le nom de la variable contenant *a*.

Chaque affectation est réalisée par la fonction *affectation – out* pour le cas de multiplications out-place et par la fonction *affectation – in* pour le cas de multiplications in-place. Si cette affectation est la dernière, la fonction rajoute une ligne "return xa;" en notant *xa* la variable contenant x^a , la valeur que la fonction générée doit retourner.

3.3.4 Génération du code c complet

La fonction *all – affectation* se charge d'appeler la fonction *affectation* sur chacun des éléments de la *alist*, en actualisant la *alist* et *variable – list* à chaque fois .

La fonction *entete – file*, comme son nom l'indique, écrit l'entête du fichier nécessaire au fonctionnement de la fonction générée dans le fichier *.c* passé en argument. C'est à dire :

```
#include <stdio.h>
#include <stdlib.h>
```

La fonction *entete – function* écrit l'entête de la fonction. Par exemple :

$$\text{int } \textit{exp3}(x)\{$$

En appelant successivement *entete – file*, *entete – function*, *all – affectation*, et (fprintf file ")) on obtient le fichier *.c* complet. C'est ce que fait la fonction *complete – file* qui prend en argument un fichier et une liste (la chaîne d'additions) qu'elle transforme en *alist* en appelant

addlist2list – of – list.

Enfin la fonction qui permet de générer tout le code c à partir d'un entier n est *global – func*. Elle prend en argument n et un entier in qui permet de choisir entre la multiplication in-place (*in* = 1) et out-place (*in* = 0). Elle ouvre un fichier, appelle la fonction *complete – file* avec en argument ce fichier et une liste qui est une chaîne d'additions générée à partir de n et referme le fichier.

3.3.5 Résultats

Sur les figures 3 et 4 on a un exemple de code c généré dans le cas de multiplications in-place (figure 3) et out-place (figure 4)

```
#include <stdio.h>
#include <stdlib.h>

int exp7(int x){
int x1 = x * 1;
int x2 = x1 * x1;
x1 *= x2;
x1 *= x2;
x2 *= x1;
return x2;
}
```

Fig. 3: Code généré en appelant
(global-func 7 '* 1)

```
#include <stdio.h>
#include <stdlib.h>

int exp7(int x){
int x1 = x * 1;
int x2 = x1 * x1;
int x3 = x1 * x2;
x1 = x2 * x3;
int x7 = x2 * x1;
return x7;
}
```

Fig. 4: Code généré en appelant
(global-func 7 '* 0)

On peut facilement tester ces fonctions pour vérifier qu'elles fonctionnent correctement, comme on le voit sur la figure 5

```
#include <stdio.h>
#include <stdlib.h>

int exp7(int x){
    int x1 = x * 1;
    int x2 = x1 * x1;
    int x3 = x1 * x2;
    x1 = x2 * x3;
    int x7 = x2 * x1;
    return x7;
}

int main(){
    printf("2 exposant 7 = %d\n", exp7(2)); //2^7 = 128
    return 0;
}
```

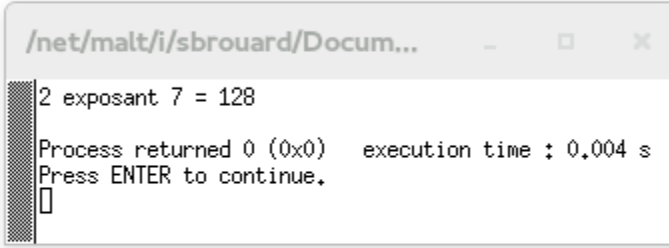


Fig. 5: Test d'une fonction générée

3.3.6 Améliorations

La fonction générée n'est en fait pas de tout à fait complexité en espace minimale, en effet, au début de la fonction la variable `x1` stocke `x`, on aurait donc faire en sorte de ne pas créer de nouvelle variable pour cela.

De plus la méthode que l'on utilise pour déterminer pour un d'une chaîne d'additions éléments de quels élément de cette même chaîne il est la somme ne nous garantis pas une utilisation d'un minimum de registre. En effet, deux couples différents d'éléments d'une chaîne d'additions peuvent donner un même nombre en étant sommés. Une amélioration dans la production de code c qui pourrait être fait serait donc de déterminer quelle suite de couple menant à l'entier `n` garanti une complexité en espace minimale par rapport à l'ensemble des fonctions d'exponentiation. Mais cela poserait des problèmes algorithmiques beaucoup plus complexes.

3.4 Calcul du nombre de registres nécessaires au calcul

Pour déterminer la qualité d'une chaîne d'additions, ce qui permettrait de choisir entre plusieurs chaînes d'additions pour un même entier `n`, on peut chercher à calculer le nombre de registres nécessaire au calcul de x^n .

La fonction **nb-registre** permet de calculer automatiquement le nombre de registres nécessaires pour calculer x^n à partir de la chaîne d'addition pour l'entier `n` que nous avons généré et de l'algorithme que nous avons utilisé.

Cette fonction reprend le principe de la génération de code mais au lieu de construire une liste ou d'écrire dans un fichier elle calcule le nombre de registre.

Ainsi à chaque fois que la fonction de génération de code génère une déclaration de nouvelle variable, **nb-registre** incrémente de 1 le nombre de registres utilisés.

4 Conclusion

Le langage `scheme` permet de générer du code facilement réutilisable : on a juste à changer l'opérateur pour générer une fonction d'exponentiation, une fonction de multiplication, ou autre, ou changer `n` pour changer d'exposant. Ce qui permet de ne pas à avoir à réécrire une fonction pour chaque cas particulier et d'automatiser des calculs.

Les chaînes d'additions permettent quand à elles de trouver des algorithmes efficaces d'exponentiation. Ces algorithmes peuvent de plus être optimisés en gérant les variable de telle sorte à pouvoir réutiliser certains registres.