Compilation
Année 2006-2007
4 avril

Florent Bouchez
Christoph Lauter

Partiel de Compilation
Version avec solutions

1 Saoûls comme des polonais

Le but de cet exercice est de donner le code `ocamllex` et `ocamlyacc` pour une calculette en notation polonaise inverse.

La notation polonaise inverse ou bien notation postfixée est une façon d'écrire des expressions avec un parenthésage implicite. Bien qu'inventée très tôt et employée anecdotiquement dans les calculettes des années 70, elle reste vivante dans plusieurs langages comme par exemple le Postscript.

Pour écrire une expression faisant appel à un opérateur $f : D^n \to D^m$, on donne d'abord n expressions qui deviendront les opérandes de l'opérateur f donné ensuite. Son évaluation produira m valeurs qui pourront servir comme arguments au prochain opérateur. Voici quelques exemples avec leur traduction en notation infixée habituelle :

$$\begin{aligned} 4\ 5\ + &\sim 4 + 5 \\ 4\ 5 + \exp &\sim \exp(4 + 5) \\ 4\ 5\ +6\ 3\ +\ \times &\sim (4 + 5) \times (6 + 3) \end{aligned}$$

Exercice 1 : Parfaitement à jeûn, je la vois qui bascule[1]

Concentrons-nous d'abord sur l'analyse lexicale, en particulier sur l'analyse de nombres réels écrits en notation scientifique. Un nombre en notation scientifique est composé d'un signe s + ou - optionnel et d'une partie entière n, optionnellement suivi d'un point `.` et d'une partie fractionnelle f auxquels se rajoute optionnellement un facteur de mise à l'échelle. Le facteur de mise à l'échelle se compose de la lettre `e`, d'un signe s' + ou - optionnel et d'un entier E. La valeur associée à un tel nombre est $s \cdot (n + f) \cdot 10^{s' \cdot E}$. À titre d'illustration, voici quelques exemples :

```
4.3
-5.2e-5
+12.07
-0.0e-0
3e10
```

Question 1.1. Montrez que le langage formé par les nombres écrit en notation scientifique est régulier en donnant une expression regulière qui l'engendre.

Question 1.2. Donnez un automate fini qui reconnaît le même langage. Minimisez ensuite votre automate pour qu'il ait le nombre minimal d'états. Servez-vous du procédé vu en cours.

Question 1.3. En compilation, on se sert d'automates finis pour l'analyse lexicale. Quelle est la fonctionalité supplémentaire que doit implémenter un analyseur lexical comme celui produit par `ocamllex` ? Traditionellement elle n'est pas présente dans la représentation théorique d'automates finis. Pensez aux "valeurs renvoyées" par l'automate et au fait qu'il y a reconnaissance d'une suite de lexèmes.

[1] la virgule flottante

Exercice 2 : Analyse syntaxique des expressions

Au premier abord, le langage des expressions en notation polonaise inverse ressemble à une suite finie de constantes et d'opérateurs. Montrez que cette suite forme un langage régulier. Convainquez ensuite le correcteur que le langage des expressions en notation polonaise inverse ne forme pas un langage régulier. Donnez des exemples de suites reconnues par un automate fini qui ne sont pas des expressions bien-formées du langage. Quelle est la structure de données dont vous avez besoin en plus pour ne reconnaître que les expressions bien-formées ? Faudra-t-il sortir les gros moyens comme `ocamlyacc` pour notre calculette ?

Exercice 3 : Germination des graines de calculettes

Remplissez les lignes marquées `A COMPLETER` pour compléter le fichier `ocamllex` de la feuille annexe. La fonction `evaluate` analysera une chaîne de caractères prise en argument, évaluera l'expression en notation polonaise inverse et renverra le ou les résultat(s) du calcul. Si une chaîne de caractère ne représente pas une expression correcte, une exception sera levée.

2 Mmmh ? heu... où suis-je ?

Exercice 4 : Bien gérer ses petites cases

Question 4.1. Pour chacune des variables a, b, c, d, e de la fonction `dummy` suivante, écrite en Pascal, la garderiez vous dans un registre ou en mémoire ? Justifiez vos choix.

```
function dummy(a : integer; var b : integer) :integer;
var c : array[1..3] of integer;
  d, e : integer;
begin
  d := a + b;
  e := stupid(c, b);
  return (e + c[d]);
end;
```

```
function stupid(x : array of integer; var y : integer) : integer;
...
```

Note : les fonctions sont similaires aux procédures, mais ont une valeur de retour dont le type est défini par la première ligne de la déclaration, par un « `: <type> ;` » après les arguments de la fonction et dont la valeur est fixée par l'instruction `return`.

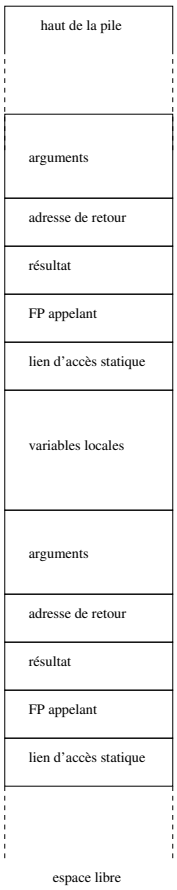
Solution: *En fait de nombreuses réponses étaient possibles, à condition de bien argumenter. Nous proposons celle-ci : a priori, ne connaissant rien sur la fonction* `stupid`*, on suppose qu'elle écrase tous les registres. Alors :*

a Reg, car meurt à la définition de d ;
b Mem, car modifiable par `stupid` *;*
c Mem, car tableau, et utilisée après appel ;
d Mem, car vivante pendant l'appel à `stupid` *;*
e Reg, car definie par l'appel et meurt à la fin de la fonction.

Exercice 5 : Les rechargables sont meilleures pour l'environnement !

On utilisera la représentation des *cadres* de la pile présentée sur la partie gauche de la figure 1. À titre de rappel, le *FP appelant* désigne le pointeur vers le début du cadre de la fonction appelante, et le *lien d'accès statique* est un pointeur vers le cadre de la fonction lexicalement englobante.

Question 5.1. La partie gauche de la figure 1 représente l'état d'une partie de la pile pendant l'exécution d'un programme. Identifiez sur cette figure un cadre complet (ensemble créé par une unique fonction) et justifiez votre choix.




```
program frames;

var x : integer;

procedure A(i : integer);
  procedure B(i : integer);
    procedure C(i : integer);
    begin
      x := x + 1;
      A(i-1);                          10
    end;
  begin
    C(i);
  end;
begin
  if (i <= 0)
  then
    write(x)
  else
    B(i);                              20
end;

begin
  A(1);
end
.
```

FIG. 1 – Un bout de pile et un programme

Solution: *Ici, le plus important était que ce qui est toujours là ait un offset fixe dans la frame, par exemple l'adresse de retour, le résultat, etc. Aussi c'est une très mauvaise idée de mettre les arguments ou les variables locales, qui sont en nombre variable pour chaque fonction en début de frame. Dans la suite on considérera que la frame commence à l'adresse où le* `FP` *appelant est enregistré.*

À noter aussi que, comme montré sur la figure, la pile croît vers le bas (l'espace libre est en bas de la pile).

Question 5.2. Le programme `frames` (partie droite de la figure 1) est exécuté. Dessinez l'état de la pile lors de l'affichage de *x* (ligne 18 :`write(x)`). En particulier, vous indiquerez vers où pointent les liens « cadre de l'appelant » et « lien d'accès statique ».

Solution: *Le tableau ci-dessous montrent vers quels frames pointent les liens.*

Frames (de haut en bas dans la pile)	*Program*	*A(1)*	*B(1)*	*C(1)*	*A(0)*
Static link	*inconnu*	*Program*	*A(1)*	*B(1)*	*Program*
Caller's Frame	*inconnu*	*Program*	*A(1)*	*B(1)*	*C(1)*

Question 5.3. Lors de l'appel à `A` par `C`, le lien d'accès statique de `A` doit pointer vers le cadre du programme principal `frames`. Dans quelle fonction se trouvera le code assembleur qui calculera cette adresse ? Donnez une version possible de ce code.

Note : utilisez une syntaxe « à la C », avec des instructions `load` et `store` (pour charger et sauvegarder en mémoire). Les registres seront notés `r1`, `r2`, ...

Solution: *Le code assembleur devra se trouver dans la fonction C, car A ne sait pas par qui elle est appelée, donc ne connaît pas la profondeur de son appelant, or on a besoin de savoir que C est de profondeur 3 (si on considére que le programme principal est de profondeur 0) pour savoir le nombre de liens statique que l'on doit suivre avant d'arriver à la frame du programme principal.*

C sait qu'elle est de profondeur 3. Elle sait aussi que A est de profondeur 1 et donc que le SL de A devra pointer sur la frame du programme principal.

opérations	*contenu de* `r1`
`r1 := rFP`	*FP de C*
`r1 += 1`	*adresse de FP de B*
`r1 := load r1`	*FP de B*
`r1 += 1`	*adresse de FP de A*
`r1 := load r1`	*FP de A*
`r1 += 1`	*adresse de FP du programme principal*
`r1 := load r1`	*FP du programme principal*

puis pour mettre au bon endroit	*contenu de* `r2`
`r2 := rFP`	*FP de C*
`r2 += size of C`	*futur FP de A*
`r2 += 1`	*emplacement du SL de A*
`store r1 at r2`	

Question 5.4. On suppose que la variable x est stockée dans la pile. Quel serait le code généré pour exécuter l'instruction de la ligne 9 : `x := x + 1` ?

Solution:

opérations	*contenu de* `r1`
`r1 := rFP`	*FP de C*
`r1 += 1`	*adresse de FP de B*
`r1 := load r1`	*FP de B*
`r1 += 1`	*adresse de FP de A*
`r1 := load r1`	*FP de A*
`r1 += 1`	*adresse de FP du programme principal*
`r1 := load r1`	*FP du programme principal*
`r1 += 2`	*adresse de* `x`
`r2 := load r1`	*r2 contient* `x`
`r2 += r2+1`	*r2 contient* `x+1`
`store r2 at r1`	*et voilà*

Exercice 6 : Le p'tit commerce de la mémoire

Certains processeurs ont des contraintes très spécifiques. Dans cet exercice, on en considère un qui ne peut charger ou sauvegarder en mémoire que si l'adresse est dans un registre spécifique : le *registre d'adresse*, noté r_a. Par contre, celui-ci possède des version avec auto-incrément (ou auto-décrément) des instructions standard *load* et *store*. On considérera qu'il n'y a que deux registres : r_a et r. Les instructions disponibles sont les suivantes :

Instruction	Effet
$r_a \leftarrow val$	charge une valeur dans le registre d'adresse
$r \leftarrow *(r_a)$	charge dans r la valeur mémoire pointée par r_a
$r \leftarrow *(r_a{+}{+})$	idem, puis fait pointer r_a sur l'adresse suivante
$r \leftarrow *(r_a{-}\ {-})$	idem, puis fait pointer r_a sur l'adresse précédente
$*(r_a) \leftarrow r$	sauvegarde la valeur dans r dans la mémoire à l'adresse r_a
$*(r_a{+}{+}) \leftarrow r$	idem, puis fait pointer r_a sur l'adresse suivante
$*(r_a{-}\ {-}) \leftarrow r$	idem, puis fait pointer r_a sur l'adresse précédente
$r \leftarrow r + *(r_a)$	incrémente r de la valeur mémoire pointée par r_a
$r \leftarrow r + *(r_a{+}{+})$	idem, puis fait pointer r_a sur l'adresse suivante
$r \leftarrow r + *(r_a{-}\ {-})$	idem, puis fait pointer r_a sur l'adresse précédente

Exemple : assigner b à a puis charger c dans le registre r :

a, b, c quelconques	a, b, c consécutifs dans la pile
$r_a \leftarrow @a$	$r_a \leftarrow @a$
$r \leftarrow *(r_a)$	$r \leftarrow *(r_a{-}\ {-})$
$r_a \leftarrow @b$	$*(r_a{-}\ {-}) \leftarrow r$
$*(r_a) \leftarrow r$	$r \leftarrow *(r_a)$
$r_a \leftarrow @c$	
$r \leftarrow *(r_a)$	

Pour simplifier, on supposera que chaque opération coûte 1. Pour l'exemple ci-dessus, il est donc possible d'économiser un tiers des opérations par un arrangement judicieux des données.

Le problème général s'énonce alors comme suit : étant donnée une suite d'opérations, placer les variables utilisées dans la pile de manière à minimiser le nombre total d'instructions nécessaires pour réaliser ces opérations.

On utilisera par la suite l'exemple d'opérations suivant :

$$\begin{aligned} c &\leftarrow a + b \\ f &\leftarrow d + e \\ a &\leftarrow a + d \\ c &\leftarrow d + a \\ b &\leftarrow d + f + a \end{aligned}$$

La *trace* des accès aux variables est la liste des accès aux variables dans l'ordre d'exécution du programme ; par exemple $a, b, c, a, \ldots$

Question 6.1. On considère que dans l'opération $x + y$, on accède à x avant y. Quelle est alors la trace des accès aux variables pour l'exemple ci-dessus ?

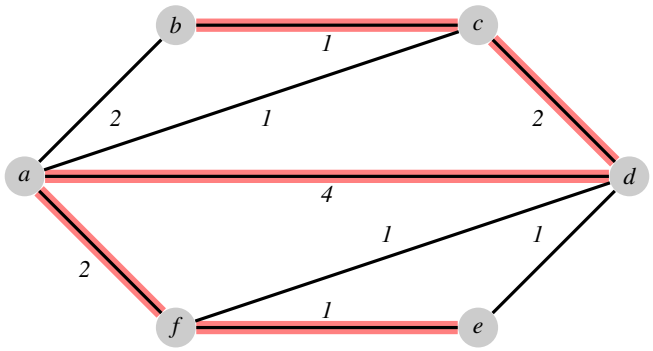
Solution: $a, b, c, d, e, f, a, d, a, d, a, c, d, f, a, b$

Soit un placement en mémoire de n variables. Une trace $T = x_1, x_2, \ldots, x_m$ a un coût $c(T) = \sum_{i=1}^{m-1} N(x_i, x_{i+1})$ où $N(x, y)$ vaut 0 si x et y sont voisins en mémoire, 1 sinon.

Question 6.2. On cherche à trouver le placement en mémoire qui minimise $c(T)$. Proposez une modélisation de ce problème à l'aide d'un graphe et trouvez une solution optimale pour l'exemple.

Solution: *Une clique avec un sommet par variable. Poids pour une arête égal au nombre de fois que ces sommets sont adjacents dans la trace. Sur la figure, on n'a pas représenté les arêtes de poids nul. La ligne surlignée correspond*

à un chemin de poids maximal : (e, f, a, d, c, b)*. Le coût de ce placement en mémoire est la somme des arêtes non choisies :* $ab + ac + fd + ed = 5$.



Question (difficile?) 6.3. Trouvez un algorithme polynomial optimal (à prouver) pour ce problème ou prouvez qu'il est NP-complet.

Solution: *Probablement NP-complet avec réduction au voyageur de commerce (il faudra alors maximiser le poids du chemin). Mais attention ce n'est pas si trivial.*

En effet le graphe n'est pas quelconque. Donc pour une instance du TSP, avec par exemple une arête de xy de poids w, il faut créer par exemple une instruction `P := x+y+x+y+... +x`*. On peut créer ainsi une suite d'assignations à* `P`*, ce qui rajoute un élément à notre clique.*

Pour que `P` *ne gêne pas, il faut que les poids des arêtes qui le relient à chacune des autres variables soient les même, donc à la fin de la construction, on rajoute par exemple* `P :=x+P+x+...` *autant qu'il faut pour que tous soit égaux. Pour des raisons de parité, il peut rester une différence de 1 entre Px et Py pour x et y quelconques. Pour que cette différence ne gêne pas le choix du départ du TSP, il suffit de doubler les poids de toutes les autres arêtes de la clique, ainsi les poids sont tous pairs, et une différence de 1 n'influera sur le départ du TSP. Donc un parcours max à l'intérieur de la clique sauf P donnera bien une solution à notre problème et inversement.*

On considère maintenant que le placement en mémoire est fixe. En revanche l'ordre dans lequel sont effectuées les additions d'une opération est à la discrétion du compilateur.

Question 6.4. Montrer que pour un ensemble d'opérations donné, les différentes traces possibles sont exactement de la forme $x_1, Y_1, x_2, Y_2, \cdots Y_{k-1}, x_k$, avec x_i variables fixées et les Y_i sont des séquences de variables où toute permutation est possible.

Montrer que l'on peut se ramener au cas où pour tout i, pour tout j, x_j apparaît au plus une fois dans Y_i sauf x_i et x_{i+1} qui n'y apparaissent pas.

Solution: *Il y a une erreur dans l'énoncé : il n'y a pas de* x_1 *puisque les* x_i *correspondent aux variables définies (membres gauche des affectations). Ces variables apparaissent toujours à des endroits fixes puisqu'une fois un calcul terminé, il faut obligatoirement le stocker dans la variable définie. Par contre, on est libre d'effectuer les additions dans l'ordre qu'on veut puisque cette opération est commutative. En particulier, on peut regrouper ensembles les variables identiques (on n'a alors pas besoin de déplacer le* r_a*), et on peut aussi en profiter pour mettre les opérandes* x_i *au début (*x_i *est le résultat de l'opération précédente, et donc* r_a *est bien placé au début du calcul), et les opérandes* x_{i+1} *à la fin (il faudra de toutes façons déplacer le* r_a *sur cette case). donc* x_i *et* x_{i+1} *n'apparaissent pas, et on peut permuter les autres opérande comme on veut.*

Question 6.5. Trouvez un algorithme polynomial optimal (à prouver) pour ce problème ou prouvez qu'il est NP-complet.

Solution:*Le problème ici est simple : il suffit pour chaque groupe X_i d'ordonner les éléments de la même manière que l'ordre défini par le placement en mémoire. L'algorithme suivant donne l'idée générale (il faut un peu plus de tests sur le choix des voisins de x_i et x_{i+1}) :*

Algorithm 1: Permutation des opérandes

soit $O = y_1, \cdots, y_n$ l'ordre en mémoire
pour chaque *i de* 1 *à* $m - 1$ **faire**
  ordonner X_i selon O
  si $\exists x$ *voisin de* x_i *dans* X_i *qui n'a qu'un voisin dans* X_i **alors**
    ordonner x en premier et le faire suivre par la chaine éventuelle à laquelle il était relié
  fin
  si $\exists x$ *voisin* x_{i+1} *dans* X_i *qui n'a qu'un voisin dans* X_i **alors**
    ordonner x en dernier et le faire précéder par la chaine éventuelle à laquelle il était relié
  fin
fin

Question (difficile?) 6.6. Que se passe-t'il si on s'autorise à modifier l'ordre des additions dans une opération *avant* de décider le placement en mémoire ? Faites des propositions.

Solution: *Celle-là c'est vraiment une question difficile. Je n'en ai aucune idée.*