

École Normale Supérieure de Lyon
Département de Mathématiques et Informatique
Master de Mathématiques et applications, parcours approfondi

Travaux dirigés & travaux pratiques du cours de compilation
Année 2004-2005

Fabrice RASTELLO

Tanguy RISSET

<Fabrice.Rastello@ens-lyon.fr>

Lyon, 31 octobre 2005

Table des matières

TD 1	Intro ; pub pour Marianne Delorme	2
TD 2	Analyse syntaxique récursive et LL(1)	9
TD 3	Analyse syntaxique LR	15
TD 4	Grammaires à attributs	21
TD 5	Correction du partiel 2003-2004	27
TD 6	Optimisations de compilation	37
TD 7	Élimination de copies redondantes	43
TD 8	Static Single Assignment	50
TD 9	Flex (TP)	53
TD 10	Bison (TP)	61
TD 11	Spim (TP)	68
TD 12	Correction de l'examen 2003-2004	76

Intro ; pub pour Marianne Delorme

La correction de ce TD à été faite par Xavier Roche, à partir du corrigé partiel de Florent de Dinechin partiellement modifié et complété par Fabrice Rastello.

1 Intro : analyse lexicale, analyse syntaxique

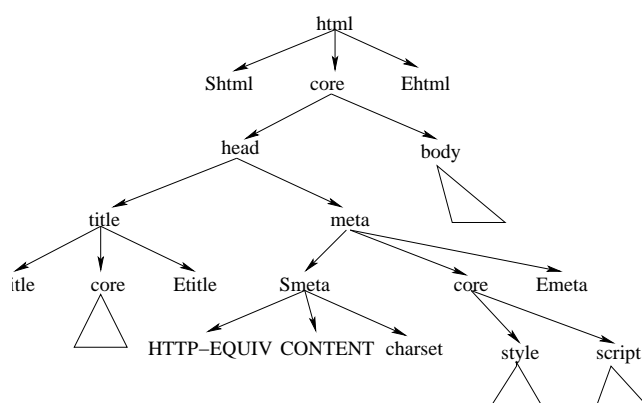
Question 1-1

Listez les tokens qui sortent de l'analyseur lexical de votre malheureux navigateur.

Les tokens sont les briques de base qui sortent de l'analyseur lexical. On les donne à manger ensuite à l'analyseur syntaxique. Il y a plusieurs réponses possibles à cette question. On peut imaginer la réponse : `<html>`, `<head>`, `<title>`, `Ecole Normale Sup\'`, `erieure de Lyon, ...`. Reste que les champs comme `meta` ou `script` posent problème car leur délimiteurs contiennent des informations (`LANGUAGE="JavaScript"` pour `script`). De plus l'utilisation des caractères `<` et `>` dans le Javascript par exemple peut éventuellement poser problème.

Question 1-2

Dessinez un arbre syntaxique (sans remplir les feuilles) pour ce code.



Question 1-3

La frontière que vous avez mise entre analyses lexicale et syntaxique est-elle univoque ? Quelle est l'influence des choix faits sur l'efficacité du compilateur ? Sur sa simplicité ?

On aurait pu répondre à la question 1-1 en choisissant une granularité plus fine avec des tokens du type `<,html,>`, ...
Un tel choix permet de définir un langage plus général tel que le xml. En contre-partie, donner

plus de boulot à l'analyseur syntaxique, c'est potentiellement augmenter la complexité du parseur (l'analyseur lexical est linéaire alors que l'analyseur syntaxique est combinatoire).

Question 1-4

L'analyseur lexical et l'analyseur syntaxique font presque la même chose mais pas tout-à-fait. Quelles sont les différences ?

Tous ce qui est faisable par un analyseur lexical (expression régulière) l'est par un analyseur syntaxique (grammaire). Mais la réciproque est fausse.

2 Automates et langages formels

Question 2-1

Qu'est-ce qu'une expression régulière sur Σ , une description régulière, une grammaire BNF, une grammaire EBNF ?

- Expressions régulières (page 10 du GBJL) :
Les expressions régulières ont une définition inductive à partir d'un alphabet Σ , elles sont équivalentes à des automates finis. On note R l'ensemble des expressions régulières sur Σ , cet ensemble est clos par union, concaténation et par opération de Kleene :

$$\left\{ \begin{array}{l} \Sigma \cup \{\epsilon\} \in R \\ \forall A \in R, \forall B \in R, A \cup B \in R \\ \forall A \in R, \forall B \in R, AB \in R \\ \forall A \in R, A^* \in R \end{array} \right.$$

- Format de grammaire BNF :
Il s'agit d'ensembles de règles de dérivation de la forme $A \rightarrow \alpha AB\beta$. Lorsqu'une même classe syntaxique apparaît dans plusieurs règles de dérivation :

$$\left\{ \begin{array}{l} A \rightarrow \alpha \\ A \rightarrow \beta \\ A \rightarrow \gamma \end{array} \right.$$

On note alors $A \rightarrow \alpha|\beta|\gamma$.

- Format de grammaire EBNF :
Les grammaires EBNF sont des grammaires BNF auxquelles on a ajouté les opérateurs '?' (0 ou une fois), '+' (répété $n > 0$ fois) et '*' (répété $n \geq 0$ fois).
- Descriptions régulières (page 34 du GBJL) :
Les description régulières sont des suites de règles de dérivation satisfaisants le format de grammaire EBNF et dont les règles de constructions n'utilisent que des classes syntaxique précédemment dérivé.
Il n'y a donc pas de cycles de la forme :

$$\left\{ \begin{array}{l} A \rightarrow \alpha...|B \\ B \rightarrow A \end{array} \right.$$

Extrait de http://en.wikipedia.org/wiki/Context-free_grammar : In linguistics and computer science, a context-free grammar (CFG) is a formal grammar in which every production rule is of the form

$$V \rightarrow w$$

BNF (Backus-Naur Form also known as Backus normal form) is the most common notation (metasyntax) used to express context-free grammars. BNF was originally named after John Backus and later (at the suggestion of Donald Knuth) also after Peter Naur, two pioneers in computer science, namely in the art of compiler design, as part of creating the rules for Algol 60.

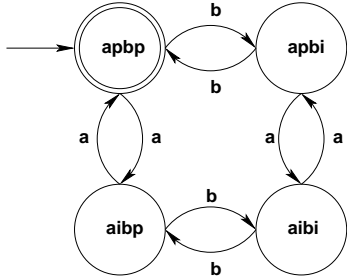
Question 2-2

Sur l'alphabet $\Sigma = \{a, b\}$, donner un automate fini déterministe reconnaissant les langages suivants :

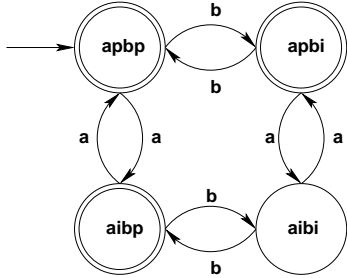
- tous les mots contenant un nombre pair de “a” et un nombre pair de “b” ;
- tous les mots contenant un nombre pair de “a” ou un nombre pair de “b” ;
- tous les mots qui n'ont pas plus (\geq) de quatre “a” consécutifs ;
- le langage $L = \{a^n b^p, n = p \bmod 3, n, p \geq 0\}$.

On note *nombre pair de a* : ap ; *nombre pair de b* : bp ; *nombre impair de a* : ai ; *nombre impair de b* : bi ;

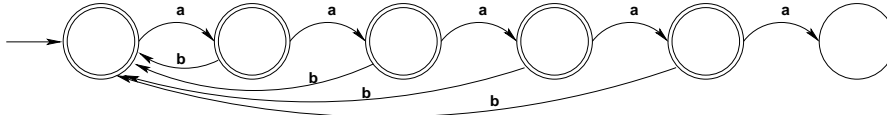
- tous les mots contenant un nombre pair de “a” et un nombre pair de “b” :



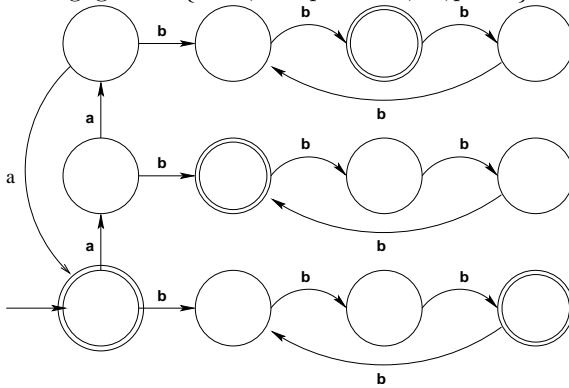
- tous les mots contenant un nombre pair de “a” ou un nombre pair de “b” :



- tous les mots qui n'ont pas plus de quatre “a” consécutifs :



- le langage $L = \{a^n b^p, n = p \bmod 3, n, p \geq 0\}$:



Question 2-3

Donnez, si vous pouvez, des expressions régulières pour les langages précédents.

Dans l'ordre :

$$\begin{cases} (aa|bb|(ab|ba)((aa|bb))^*(ab|ba))^* \\ (b^*ab^*ab^*)^*|(a^*ba^*ba^*)^* \\ (a|aa|aaa)(bab^*|baab^*|baaab^*) \\ (aaa)^*(ab|aabb|(bbb))^* \end{cases}$$

Question 2-4

Montrez que le langage $L = \{a^n b^n | n \geq 1\}$ ne peut pas être défini par une expression régulière.

C'est une application directe du lemme de l'étoile qui exploite le caractère fini du nombre d'états de l'automate correspondant à une expression régulière. On trouve par exemple dans <http://www.liafa.jussieu.fr/~carton/Enseignement/Complexite/MMFAI/Cours/pumping.html> :

Lemme *Pour tout langage rationnel L , il existe un entier n tel que tout mot f de longueur supérieur à n se factorise $f = uvw$ où le xuv^*wy est inclus dans L .*

On montre facilement que le langage $L = \{a^n b^n | n \geq 0\}$ ne satisfait pas la propriété du lemme ci-dessus. On en déduit qu'il n'est pas rationnel. Par contre le langage $L' = L + A^*baA^*$ satisfait la propriété bien qu'il ne soit pas rationnel.

On peut aussi faire une preuve directe (cela dit la preuve du lemme de l'étoile est triviale et du même genre) : d'abord, si une expression régulière définit le langage L , alors il existe un automate *fini* déterministe vérifiant L . Nous allons raisonner par l'absurde sur le caractère fini de cet automate :

Soit A un automate déterministe vérifiant L . On note e_n l'état de A obtenu après les transitions $a^n b$. De cet état e_n , on ne peut arriver sur un état final qu'en faisant les transitions b^{n-1} (et non b^n ou b^{n-2} ...). Ainsi, chaque e_n représente un état différent de A . L'automate A possède un nombre infini d'états e_1, e_2, \dots . Ce qui est absurde.

Question 2-5

Donnez une description régulière qui décrit toutes les chaînes composées de chiffres sans deux fois le même consécutivement. Essayer aussi sous forme d'une expression régulière.

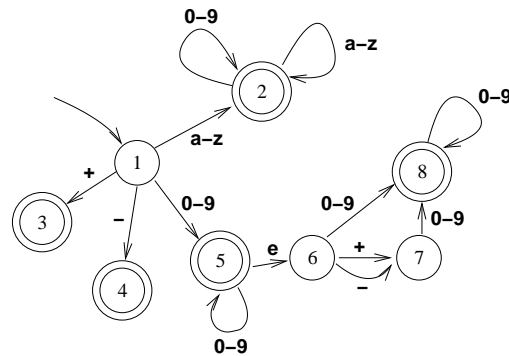
Une description régulière possible serait :

$$\begin{aligned} d_0 &\rightarrow 0 \\ d_1 &\rightarrow (d_0|\epsilon)(1d_0)^*(1|\epsilon) \\ d_2 &\rightarrow (d_1|\epsilon)(2d_1)^*(2|\epsilon) \\ d_3 &\rightarrow (d_2|\epsilon)(3d_2)^*(3|\epsilon) \\ d_4 &\rightarrow (d_3|\epsilon)(4d_3)^*(4|\epsilon) \\ d_5 &\rightarrow (d_4|\epsilon)(5d_4)^*(5|\epsilon) \\ d_6 &\rightarrow (d_5|\epsilon)(6d_5)^*(6|\epsilon) \\ d_7 &\rightarrow (d_6|\epsilon)(7d_6)^*(7|\epsilon) \\ d_8 &\rightarrow (d_7|\epsilon)(8d_7)^*(8|\epsilon) \\ d_9 &\rightarrow (d_8|\epsilon)(9d_8)^*(9|\epsilon) \\ S &\rightarrow d_9 \end{aligned}$$

L'expression régulière correspondant peut être obtenue en dérivant S en d_9 , puis en $(d_8|\epsilon)(9d_8)^*(9|\epsilon)$, puis chaque d_8 en $(d_7|\epsilon)(8d_7)^*(8|\epsilon)$ ce qui donnerait $((d_7|\epsilon)(8d_7)^*(8|\epsilon)|\epsilon)(9(d_7|\epsilon)(8d_7)^*(8|\epsilon))^*(9|\epsilon)$, puis chaque d_7 en $(d_6|\epsilon)(7d_6)^*(7|\epsilon)$, etc. On voit que la taille de l'expression croît exponentiellement (fois 2 à chaque dérivation), de telle manière que l'expression finale aurait une taille de l'ordre de $\Omega(2^9)$. Cela motive l'utilisation de descriptions régulières, même pour représenter des expressions régulières.

Question 2-6

Voici un joli automate



Il reconnaît quoi ? (indice : enfin un exemple qui sert à quelquechose...) Donnez-en une expression régulière.

Il reconnaît tous les lexèmes utiles pour écrire des expressions à deux opérations sur des nombres flottants à mantisse entière : c'est à dire les entiers, les noms de variables, les flottants et les opérateurs '+' et '-'.

Question 2-7

Si on l'utilise pour trouver des lexèmes dans un fichier d'entrée, combien de caractères doit on examiner au pire après la fin d'un lexème pour pouvoir l'identifier ?

3. C'est l'occasion de rappeler que les automates qu'on construit reconnaissent le plus long préfixe qui matche l'entrée. En voyant arriver 12e+345, l'analyseur hésite entre 12/e/+xxx et un nombre flottant jusqu'à avoir mangé le 3. Après, il est certain qu'il pourra de toutes manières reconnaître un nombre flottant.

Question 2-8

En général un analyseur lexical reconnaît le plus long préfixe. Comment traiter le cas des commentaires de manière à ne pas oublier du code dans l'exemple ci-dessous en Caml ?

```

let f x = (* une fonction bidon *)
  if x mod 2 = 0 then x/2 else x+3;;
  (* fin de la fonction bidon *)

```

Le problème avec deux commentaires est que le plus long préfixe vérifiant (*...*) est l'ensemble des deux codes : le commentaire englobe alors l'instruction :

```

if x mod 2 = 0 then x/2 else x+3;;

```

De plus on ne peut borner la taille des commentaires, on s'en sort généralement en traitant les commentaires par une première passe.

3 Des expressions régulières aux automates

Question 3-1

Soit R une expression régulière sur l'alphabet Σ . Après avoir complété Σ avec un symbole ϵ signifiant "vide", donnez une méthode simple de construction d'un automate reconnaissant R .

On sait construire les automates (triviaux) qui reconnaissent un caractère de Σ , ainsi que le $.$ (wildcard).

Si les expressions régulières R_1 et R_2 sont reconnues respectivement par les automates A_1 et A_2 , alors

- Un automate reconnaissant $R_1 R_2$ est construit comme la juxtaposition de A_1 et A_2 . Il a pour entrée l'état d'entrée de A_1 , pour états accepteurs les états accepteurs de A_2 , et possède des ϵ -transitions de tous les états accepteurs A_1 vers l'entrée de A_2 .

- Un automate non déterministe reconnaissant $R_1|R_2$ est construit de la manière suivante à partir des automates A_1 et A_2 : son état d'entrée est un nouvel état qui possède deux ϵ -transitions vers les états d'entrée de A_1 et A_2 . Il possède un état accepteur, qui est un nouvel état recevant des ϵ -transitions de tous les états accepteurs de A_1 et A_2 .
- On construit facilement de même un automate reconnaissant R_1^* à partir de A_1 en reliant ses sorties à son entrée par des ϵ -transitions.

Question 3-2

Bornez le nombre d'états de l'automate construit.

Par induction on voit que le nombre d'états est linéaire en le nombre de caractères de l'expression régulière. En effet, les automates reconnaissant un caractère ont deux états, l'opération de concaténation et l'opérateur de Kleene ne créent aucun état et l'union construit deux nouveaux états : si l'on nomme n le nombre de caractères, l'automate construit contiendra au plus $2n$ états.

Question 3-3

Cet automate est-il déterministe ou non déterministe ?

Cet automate est clairement non déterministe (utilisation d' ϵ -transitions).

Question 3-4

Rappelez le principe de la "déterminisation" d'un automate fini non déterministe. Appliquez-le à un automate reconnaissant une expression régulière. En principe cela doit commencer à ressembler à la méthode vue en cours. Et si vous trouvez que cela ne ressemble pas, pas de panique, on fera pour l'analyse syntaxique un TD sur la méthode des boulettes.

Le principe est de parcourir l'automate non déterministe et de construire un nouvel automate déterministe, dans lequel chaque état représente un ensemble d'états de l'AFN, l'ensemble des états accessibles après avoir lu les mêmes symboles d'entrée.

À partir d'un automate non déterministe $A_1(N, \Sigma, \delta_N, n_0, N_F)$, on crée l'automate déterministe $A_2 = (D, \Sigma, \delta_D, d_0, D_F)$ tel que d_0 soit l'ensemble des états atteignables par ϵ -transition à partir des états initiaux de n_0 . Lorsque l'on réalise une transition de l'ensemble E , état de A_2 , selon le caractère a , on obtient l'ensemble des états de A_1 atteignables par a à partir des états de E . Cet ensemble est clos par ϵ -transition.

Les états acceptants de A_2 sont les ensembles contenant au moins un état final de A_1 .

Le nombre d'états peut exploser jusqu'à 2^n où n est le nombre d'états de l'automate non déterministe.

Analyse syntaxique récursive et LL(1)

La correction de ce TD à été faite par Matthieu Gallet et Veronika Rehn à partir du corrigé partiel de Florent de Dinechin partiellement modifié et complété par Fabrice Rastello. Il reste de nombreuses typos et tout n'a pas été vérifié!

4 Grammaires

Question 4-1

Rappelez la différence entre une grammaire et une description régulière (ou ensemble de définitions-régulières?).

Une grammaire formelle, ou simplement grammaire, est formée d'un ensemble fini de symboles terminaux (qui sont les lettres ou les mots du langage), d'un ensemble fini de non-terminaux, d'un ensemble de productions dont les membres gauche et droits sont des mots formés de terminaux et de non-terminaux, et d'un axiome. Appliquer une production consiste à remplacer son membre de gauche par son membre de droite; l'application successive d'un certain nombre de productions s'appelle une dérivation. Le langage défini par une grammaire rationnelle est l'ensemble des mots formés uniquement de symboles terminaux qui peuvent être atteints par dérivation à partir de l'axiome.

Une description régulière est une grammaire sans récursivité, comme une grammaire "context-free" EBNF, mais avec la restriction que les Non-Terminaux ne peuvent pas être utilisés avant d'avoir été complètement définis. On peut toujours substituer la partie droite d'une loi dans les parties droites des lois suivantes.

Exemple de description régulière d'une grammaire reconnaissant les identifiants :

```
letter -> [a-zA-Z]
digit -> [0-9]
underscore -> _
letter_or_digit -> letter|digit
underscored_tail -> underscore letter_or_digit+
identifieur -> letter letter_or_digit* underscored_tail*
```

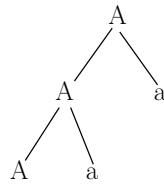
Exemple de grammaire qui n'est pas une description régulière :

```
R -> R ' | ' R | R R | R '*' | '(' R ') ' | 'a' | 'b'
```

Ce n'est pas une description régulière car cette grammaire est récursive.

Rappelez ce qu'est une grammaire récursive à gauche, récursive à droite, avec récursion directe ou indirecte. Comparez les orientations politiques de la récursivité de la grammaire, d'une part, et de l'associativité des opérateurs impliqués, d'autre part.

Exemple avec récursivité indirecte : $A \longrightarrow B\beta$, $B \longrightarrow \alpha A$


$$\begin{array}{c} + \\ \swarrow \quad \searrow \\ a \quad \quad + \\ \quad \quad \swarrow \quad \searrow \\ \quad \quad b \quad \quad c \end{array}$$

Soit la grammaire $R \rightarrow R' \mid R \mid R R \mid R *' \mid '(R)'$, $'a'$ $'b'$
 Pourquoi cette grammaire est-elle ambiguë? Proposez une grammaire équivalente non ambiguë
 donnant les priorités et associativités usuelles aux opérateurs ($*$ a la plus haute priorité, ensuite
 vient la concaténation, enfin $|$, et tous ces opérateurs sont associatifs à gauche).

Factor -> '('Exp ')', | '('Exp')' '*' | var | var'*'

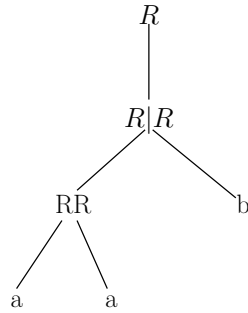


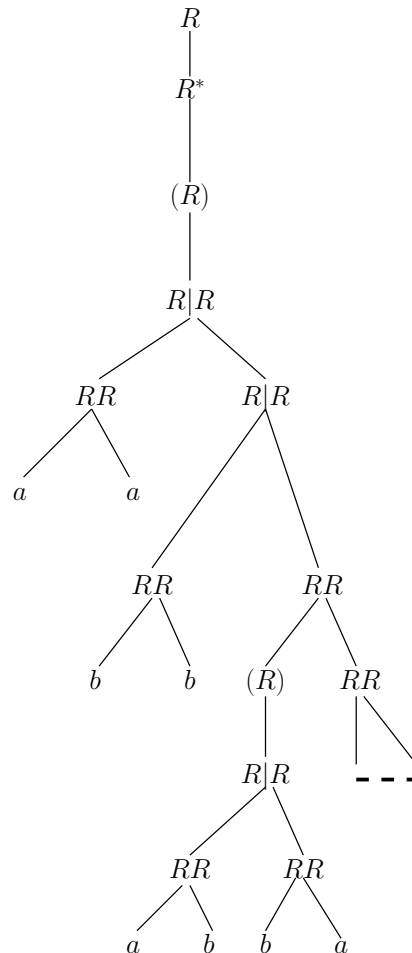
FIG. 3 – Les 2 arbres possibles pour $aa|b$

Var \rightarrow 'a' | 'b'

Question 4-4

Donnez, suivant la grammaire précédente, l'arbre syntaxique et l'arbre syntaxique réduit pour

$(aa|bb|(ab|ba)((aa|bb))^*(ab|ba))^*$



Question 4-5

Écrire une grammaire non ambiguë pour le langage des parenthèses et crochets bien équilibrés, où par convention chaque crochet fermant ferme aussi toutes les parenthèses ouvertes en suspens (entre le crochet en question et le crochet ouvrant que l'on "referme"). Exemple : $[(((\underline{[]})\underline{]})]$.

La difficulté est de permettre plus de parenthèses ouvrantes que fermantes. La première idée est d'écrire la grammaire suivante, mais celle-ci est ambiguë :

```
R -> '[' S ']' | S
S -> '(' R ')' | '(' R
```

La version suivante est non ambiguë (il y en a d'autres) :

```
R -> '[' S ']' | '(' R ')' | ''
S -> '(' S | R
```

De nos jours, ce genre de “convention” est considérée comme une Très Mauvaise Chose dans un langage de programmation...

Question 4-6

Quel est le problème avec la grammaire suivante ?

```
S -> '0' | A
A -> A B
B -> '1'
```

Décrire un algorithme éliminant dans une grammaire les productions “fautives” au sens illustré ci-dessus.

On part facilement sur une mauvaise piste en croyant que le problème est la boucle infinie et improductive dans

```
A -> A B
```

L'ambiguïté d'une grammaire étant indécidable¹, il faut renoncer à un algorithme qui détecterait et supprimerait ce type de cycles infinis ambigus, dont l'archétype est

```
A -> A | ''
```

Par contre il y a un non-terminal *inutile* : A. Formellement un non-terminal A est inutile s'il n'apparaît dans aucune dérivation donnant une séquence terminale, de type

$$L \rightarrow^* xAy \rightarrow^* xwy$$

Ceci est plus facile à détecter. Parait-il.

Pour déterminer les non-terminaux inutiles, on peut utiliser des algorithmes de type point fixe, en éliminant progressivement les termes productifs. Cette méthode revient à reconnaître des cycles dans des graphes. Pour cet exemple, on aurait :

```
S -> '0' | A
A -> A B
B -> '1'
```

```
S -> Productif
A -> A B
B -> Productif

A -> A Productif
```

A est improductif car arrivé à la dernière étape, l'algorithme ne change rien, et A n'est pas productif.

¹Florent a demandé à Marianne.

5 Analyse récursive et dérécursivation à gauche

Question 5-1

Rappelez le principe de l'analyse (grammaticale) récursive descendante. Quel rapport avec les récursions à gauche ?

L'idée de l'analyse récursive est de traduire directement les productions de la grammaire en des fonctions booléennes qui s'appellent récursivement : la concaténation devient un *et logique*, l'alternative un *ou logique*, les symboles terminaux renvoient *vrai* en consommant un token, ou bien *faux* sans rien consommer. On peut ainsi utiliser la sémantique d'évaluation paresseuse des constructeurs booléens dans les langages impératifs.

Le rapport avec la récursion à gauche, c'est que s'il y en a elle se traduit par une boucle infinie...

Question 5-2

Donnez l'algorithme de dérécursivation à gauche d'une production du type :

$$A \rightarrow A\alpha_1 | \dots | A\alpha_n | \beta_1 | \dots | \beta_n$$

entrée : Non Terminal A

nombre PA des productions de A

sortie : nombre P'A des productions de A sans récursivité directe à gauche

- Grouper les productions de PA de façon à ce que les productions avec récursion à gauche directe soient les premières

$$A \rightarrow AP_1 | \dots | AP_m | b_1 | \dots | b_n$$

et que les $b_i \in (N \cup T)^*$, $1 \leq i \leq n$ ne commencent pas avec A.

- si $m = 0$, c'est-à-dire si PA ne contient pas des productions avec récursivité à gauche directe, mettre $P'A := PA$
- Si $m > 0$, définir P'A comme le nombre des productions suivantes

$$A \rightarrow b_1 A' | \dots | b_n A'$$

$$A' \rightarrow P_1 A' | \dots | P_m A' | \epsilon$$

- Retourner le nombre des productions PA'

Question 5-3

Construisez un algorithme qui dérécursive une grammaire récursive à gauche indirecte. Testez-le sur la grammaire suivante :

$$A \rightarrow B 'x' 'y' \mid 'x'$$

$$B \rightarrow C D$$

$$C \rightarrow A \mid 'c'$$

Algorithme général d'élimination de la récursion à gauche :

input : Grammaire hors-contexte sans ϵ $G = (T; N, P, S)$

output : Grammaire hors-contexte $G' = (T, N, P', S)$

Soit $N = \{A_1, \dots, A_n\}$

for $i = 1, \dots, n$ do

for $j = 1, \dots, i-1$ do

soient $A_j \rightarrow P_1 | \dots | P_k$ toutes les productions A_j de P.

remplacer tous les productions de la forme $A_i \rightarrow A_j f$ par les productions $A_i \rightarrow P_1 f | \dots | P_k f$

éliminer la récursion à gauche directe des productions A_i

obtenues par cet algorithme grâce à l'algorithme précédent.

Pour l'exemple donné, cela donne :

$$A \longrightarrow B \text{ 'x' 'y' } \mid \text{ 'x' }$$

$$B \longrightarrow CD$$

$$C \longrightarrow A \mid \text{ 'c' }$$

$$D \longrightarrow \text{ 'b' }$$

on transforme en

$$A \longrightarrow AD \text{ 'x' 'y' } \mid \text{ 'c' } D \text{ 'x' 'y' } \mid \text{ 'x' }$$

$$D \longrightarrow \text{ 'b' }$$

puis en

$$A \longrightarrow \text{ 'c' } D \text{ 'x' 'y' } A' \mid \text{ 'x' } A'$$

$$A' \longrightarrow D \text{ 'x' 'y' } A' \mid \epsilon$$

$$D \longrightarrow \text{ 'b' }$$

Analyse syntaxique LR

La correction de ce TD à été faite par Thomas Hugel à partir : (1) du corrigé partiel de Florent de Dinechin partiellement modifié et complété par Fabrice Rastello ; (2) de notes de cours de Tanguy Risset.

6 Analyse prédictive et factorisation à gauche

Tout ceci a peu d'intérêt pratique autre que de faire travailler les neurones, puisque `yacc` est un parseur LR. Par contre les problèmes avec LL(1) vont motiver l'analyse LR.

Question 6-1

Rappelez le principe de l'analyse prédictive (connue aussi sous le sobriquet LL(1)). Quel rapport avec la factorisation à gauche ?

L'analyse récursive a mauvaise réputation à cause du coût du backtracking qu'elle implique. L'idée de l'analyse prédictive ou LL(1) est donc de précalculer, pour chaque alternative de règle dans la grammaire, l'ensemble *First* des terminaux que peut produire cette alternative en première position. On calcule par inférence ces ensembles *First*. Ce faisant on calcule au passage des ensembles *First* pour les non-terminaux (même si aucune alternative n'est égale à ce non-terminal). La seule difficulté est de gérer les non-terminaux annulables (c.à.d. qui peuvent se réduire en ϵ).

Déterminer si un non-terminal est annulable se précalcule également très bien par induction. Dans ce cas il faut également précalculer l'ensemble *Follow* des terminaux qui peuvent suivre ce non-terminal dans l'ensemble de la grammaire. Attention, il y a un ensemble *First* par *alternative*, alors qu'il y a un ensemble *Follow* par *règle annulable*.

L'analyseur correspondant est appelé *prédictif* car il prédit (en fait décide) quelle production réduire uniquement au vu du prochain *token* : dans un état donné, il réduit l'alternative dont l'ensemble *First* contient le prochain token. S'il y a plusieurs telles alternatives, on parle de conflit *First/First*.

Enfin, l'analyseur LL(1) est un ensemble de fonctions mutuellement récursives, une par non-terminal. Ces fonctions se construisent comme suit :

- un terminal se traduit par l'action “manger un token” (et vérifier que c'est l'un de ceux prédits, sinon erreur) ;
- un non-terminal se traduit par l'appel de la fonction correspondante ;
- la concaténation dans la grammaire se traduit par une séquence d'actions du programme ;
- l'alternative se traduit par un *switch* sur le prochain token. Pour les alternatives non ϵ , on teste si ce prochain token appartient à leur ensemble *First*. Pour une alternative égale à ϵ , on teste si le prochain token appartient à l'ensemble *Follow* de la règle, auquel cas on ne fait rien (on “mange le ϵ ”).

Le rapport avec la factorisation à gauche c'est que cela ne marche que si tout est bien factorisé à gauche : lorsque deux alternatives d'une règle ont le même terminal en première position, l'analyseur ainsi construit ne sait décider quelle alternative suivre. On appelle cela un conflit *first/first*. La solution est de factoriser ce terminal dans les deux alternatives.

Question 6-2

Au fait, et les grammaires récursives à gauche, elles vont ou elles vont pas ?

Non : on a un autre risque de conflit dit *First/Follow*, lorsque l'une des fonctions de l'analyseur ne peut pas décider quelle branche prendre. Un exemple est la grammaire :

$$\begin{aligned} S &\rightarrow A a b \\ A &\rightarrow A \mid \epsilon \end{aligned}$$

Calculons les ensembles *First+* et *Follow* :

	<i>First</i>	<i>Follow</i>	<i>First+</i>
S	{a}	{}	{a}
A	{ ϵ }	{a}	{ ϵ , a}

Le problème est lié à la fonction de l'analyseur associée à la règle $A \rightarrow A \mid \epsilon$. Formellement, il faudrait que $First+(A) \cap Follow(A) = \emptyset$, ce qui n'est pas le cas. En pratique, lorsque l'analyseur est sur un A et qu'il lit un a , il ne saurait pas s'il faut dériver A en A ou en ϵ car à la fois $a \in First+(A)$ et $a \in Follow(A)$.

La morale c'est que la construction automatique de l'analyseur fonctionne uniquement pour certaines grammaires, celles qui n'ont pas de conflit LL(1), et que l'on appelle donc des grammaires LL(1). Étant donné une grammaire, il est souvent (pas toujours) possible de la bricoler pour qu'elle devienne LL(1).

Question 6-3

Quel est le problème avec la grammaire suivante des `if` en C ?

```

IfElseStatement -> 'if' '(' Expression ')' Statement ElseTailOpt
ElseTailOpt     -> 'else' Statement | ''
Statement       -> ... | IfElseStatement | ...
    
```

Une solution dans ce cas est d'exprimer une priorité dans la grammaire pour résoudre le conflit LL(1). Expliquez.

Calculons les ensembles *First+* et *Follow* :

	<i>First</i>	<i>Follow</i>	<i>First+</i>
IfElseStatement	{if}	{else}	{if}
ElseTailOpt	{else, ϵ }	{else}	{else, ϵ }
Statement	{if, ...}	{else}	{if, ...}

Lorsque l'analyseur est sur un `ElseTailOpt` et qu'il lit un `else`, il ne saurait pas s'il faut dériver `ElseTailOpt` en `else Statement` ou en ϵ car $\text{else} \in First+(ElseTailOpt) \cap Follow(ElseTailOpt)$. On a un conflit *First/Follow*.

Ainsi cette grammaire n'est pas LL(1). D'ailleurs on pouvait remarquer qu'elle est ambiguë :

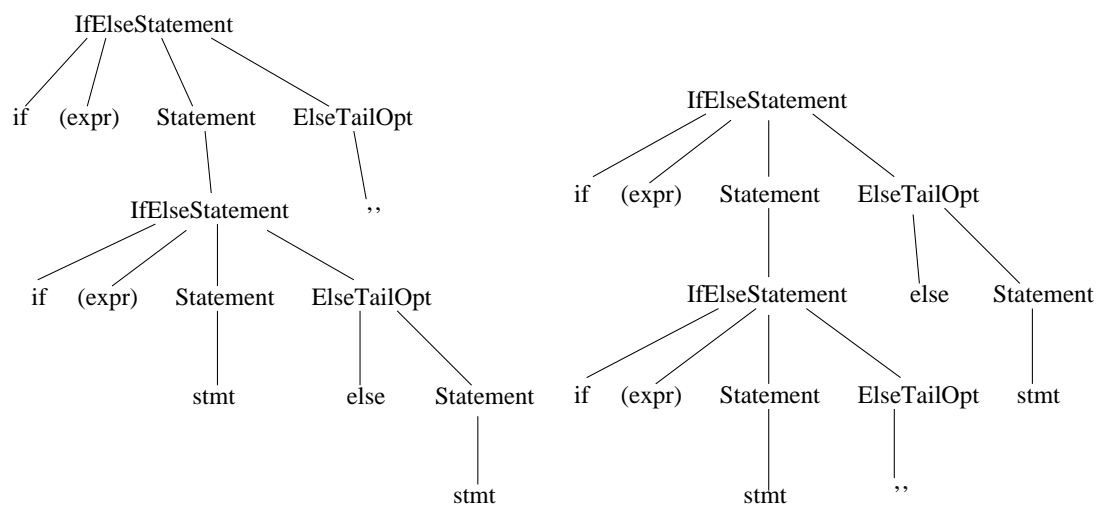


FIG. 4 – Deux arbres syntaxiques pour `if (expr) if (expr) stmt else stmt`

Voici une grammaire LL(1) qui donne la priorité au `else` :

Instruction \rightarrow WithElse | WithoutElse | Statement
 WithElse \rightarrow if '(' Expr ')' Instruction else Instruction
 WithoutElse \rightarrow '(' if Expr Instruction ')'
 Statement \rightarrow quelque chose non vide et sans if ni (au début

Cette grammaire est bien LL(1), comme on le voit en calculant les ensembles *First* (pas besoin de *Follow* car pas d' ϵ).

	<i>First</i>
Instruction	{if, (, quelque}
WithElse	{if}
WithoutElse	{(}
Statement	{quelque}

Question 6-4

Donnez la bonne vieille grammaire des expressions algébriques avec les priorités habituelles, et bricolez-là à gauche pour en faire une version LL(1).

Les expressions algébriques avec associativité à gauche :

$E \rightarrow E + T \mid E - T \mid T$
 $T \rightarrow a \mid b \mid c$

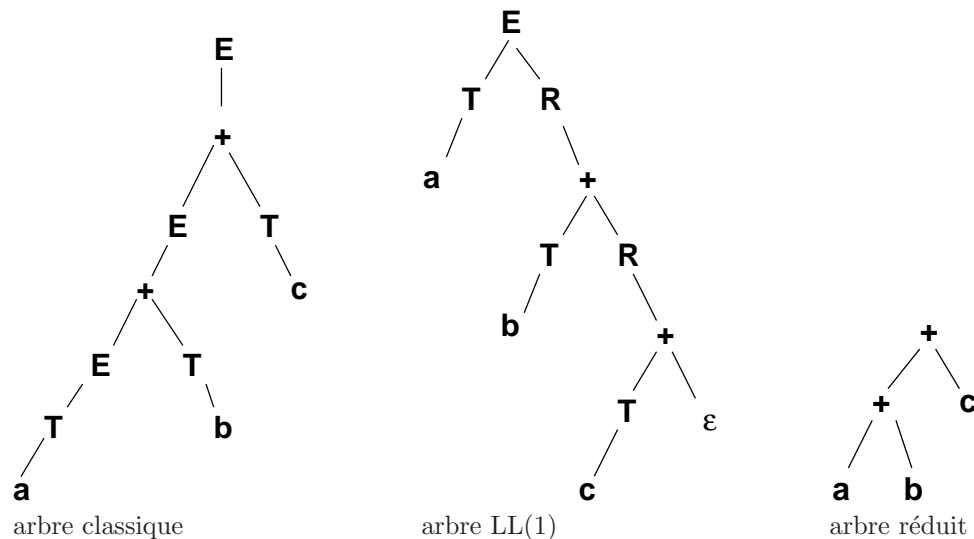
dont la version LL(1) est

$E \rightarrow T \ R$
 $R \rightarrow + \ T \ R \mid - \ T \ R \mid ''$
 $T \rightarrow a \mid b \mid c$

Question 6-5

Après avoir trafiqué notre grammaire pour qu'elle soit LL(1), on ne la reconnaît plus. Il arrive qu'elle nous construise des arbres qui ne sont plus ceux qu'on voulait. Dans quel cas ? Donnez un exemple archirépandu. Suggérez des solutions.

Pour l'exemple précédent, appliqué à $a + b + c$, les dérivations (les arbres syntaxiques étendus) sont différentes.



L'arbre LL(1) n'a pas la forme attendue car l'associativité à gauche fait naturellement pencher les arbres vers la gauche avec une récursion à gauche. Cela dit, la construction de l'arbre syntaxique réduit (l'arbre de l'expression) donnera la même chose dans les deux cas : la sémantique de chacune de ces dérivations est bien une expression avec l'associativité à gauche.

7 Questions quasi de cours

Question 7-1

Finally, the property LR(1), is a property of the language, of the grammar or of the parser?

It's first a property of the parser. Then, one defines a grammar LR(1) as being a grammar for which one can build a parser LR(1). Finally, for a given language there can be a grammar that is ambiguous (not LR(1)) and a non-ambiguous.

Question 7-2

My automaton, I don't just want it to recognize the language, I want it to build the tree. How does it work?

When one wants to build the syntax tree of the program and not just recognize the language, this is done during the *reduce* step, and the non-terminals in the stack have a pointer to the corresponding tree pieces already recognized.

Question 7-3

Why do we push onto the stack with love the states of an LR automaton, then when we reduce we throw them away without even reading them?

Because when we push (token, state) one by one, then when we pop by blocks. The top of the stack is well used in the future to remember where we have been. When pushing one does not know yet if what is pushed will be used or not, or if it will be popped without being read.

Question 7-4

Are there ambiguous LR grammars?

The answer is no. By definition, a LR (or LL, etc) grammar is a grammar that allows to reconstruct the derivation tree in a deterministic way, which implies that there must be only one. On the other hand, there exist grammars that are not ambiguous, which are neither LL nor LR. To add to the confusion, in case of conflict, it is possible to propose different ways of resolving the conflicts in the automaton (this is the third section). This does not make a grammar more LR, but it determines the automaton.

Question 7-5

What happens in case of right recursion in a LR grammar?

The stack is larger than with left recursion, because to reduce by a rule, one must have read the whole word. e.g. the example *elem* of the course.

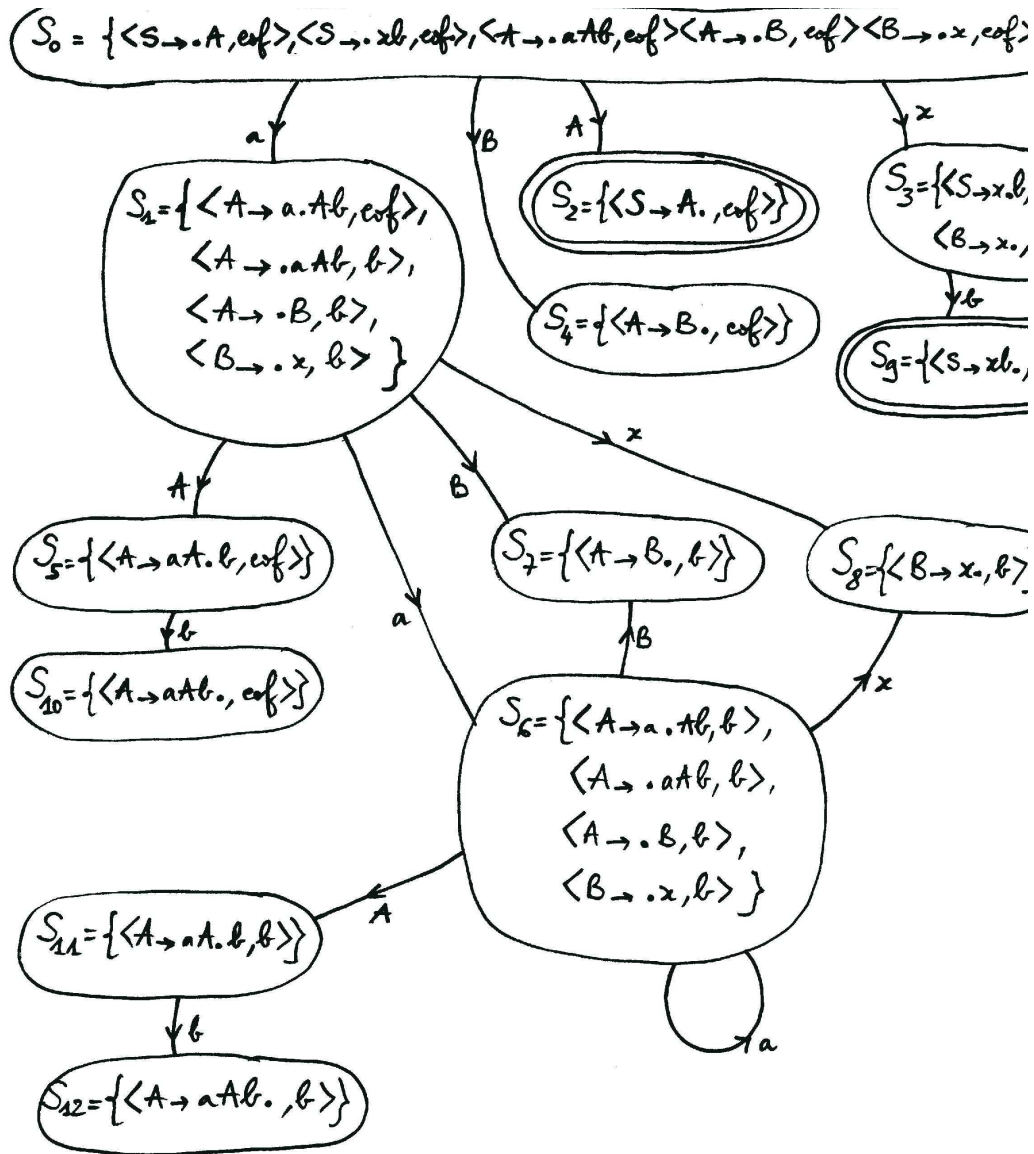
8 Moulinons un exemple

Let the grammar :

```
S -> A | xb
A -> aAb | B
B -> x
```

Question 8-1

Trouvez les tables Action, Goto et l'automate LR(1) de cette grammaire.



TD 3 - Question 3-1

Automate LR(1) de la grammaire

$$S \rightarrow A \mid x b$$

$$A \rightarrow a A b \mid B$$

$$B \rightarrow x$$

Goto	a	b	x	A	B	Action	a	b	x	eof
0	1	-	3	2	4	0	shift	-	shift	-
1	6	-	8	5	7	1	shift	-	shift	-
2	-	-	-	-	-	2	-	-	-	accept
3	-	9	-	-	-	3	-	shift	-	reduce by $B \rightarrow x$
4	-	-	-	-	-	4	-	-	-	reduce by $A \rightarrow B$
5	-	10	-	-	-	5	-	shift	-	-
6	6	-	8	11	7	6	shift	-	shift	-
7	-	-	-	-	-	7	-	reduce by $A \rightarrow B$	-	-
8	-	-	-	-	-	8	-	reduce by $B \rightarrow x$	-	-
9	-	-	-	-	-	9	-	-	-	accept
10	-	-	-	-	-	10	-	-	-	reduce by $A \rightarrow aAb$
11	-	12	-	-	-	11	-	shift	-	-
12	-	-	-	-	-	12	-	reduce by $A \rightarrow aAb$	-	-

Question 8-2

Moulinez sur axbb

pile	entrée	action
\$ S_0	axbb	shift
\$ S_0aS_1	xbb	shift
\$ $S_0aS_1xS_8$	bb	reduce by $B \rightarrow x$
\$ $S_0aS_1BS_7$	bb	reduce by $A \rightarrow B$
\$ $S_0aS_1AS_5$	bb	shift
\$ $S_0aS_1AS_5bS_{10}$	b	reject

Grammaires à attributs

La correction de ce TD à été faite par Sergueï Lenglet à partir du corrigé partiel de Fabrice Rastello. La correction n'a été que partiellement vérifiée.

9 Attributs

Question 9-1

Quelle type de grammaire à attributs est reconnue par yacc ou bison.

Yacc (ou bison) ne reconnaît que des grammaires à attributs *synthétisés* : la valeur des attributs remonte des feuilles de l'arbre syntaxique à la racine.

Question 9-2

Yacc, ne reconnaît pas les extensions EBNF (*,?,+,dots), et pour cause : comment pourrait-on étendre la notion d'attributs dans le cas d'une notation EBNF ?

Dans le cas d'une grammaire EBNF, il est nécessaire de définir une convention : comment gérer les expressions manquantes par exemple. Prenons le cas de Perl : la notation EBNF `y` est admise. Si on parse une expression de la forme `(expr1)(expr2)?(expr3)`, dans le cas où `expr2` est manquante, le deuxième attribut n'est pas vide (comme on pourrait s'y attendre), mais contient l'attribut de `expr3`.

10 Présentation du langage Alpha

ALPHA est un langage fonctionnel fortement typé. Sa caractéristique essentielle est que les objets de base qu'il manipule sont des *polyèdres convexes de valeurs*. Un polyèdre convexe est une généralisation utile de la notion de tableau multidimensionnel. On le représentera par une intersection de demi-espaces, chaque demi-espace étant défini par une inéquation affine. Voici une syntaxe typique pour un polyèdre :

```
{i,j,k | 1<=i; i<=10; 1<=j; j<=10; 1<=k; 2k+j<=i}
```

L'intérêt de cette représentation est son caractère affine, qui permet l'analyse et la transformation des domaines (donc des programmes) par des outils mathématiques bien définis (algèbre linéaire, programmation linéaire en nombre entiers). Mais ce n'est pas l'objet de ce TD.

On manipulera en fait des *ensembles finis de polyèdres convexes*, que l'on appellera désormais des *domaines*. Techniquement, l'ensemble des domaines forme un treillis pour l'inclusion, et est stable par intersection, par union, par préimage par une fonction affine et (presque) par image par une fonction affine.

Voici un exemple de programme ALPHA qui décrit un produit matrice-vecteur.

```
system matvect
  (M : {i,j | 1<=i; i<=42; 1<=j; j<=42} of real;
   V : {j | 1<=j; j<=42} of real)
  returns (R : {i | 1<=i; i<=42} of real);
```

```

var
  C : {i,j | 1<=i; i<=42; 0<=j; j<=42} of real;
let
  C = case
    {i,j| j=0} : 0.(i,j->);
    {i,j| j>0} : C.(i,j->i,j-1) + M * V.(i,j->j);
  esac;
  R = C.(i->i,42);
tel;

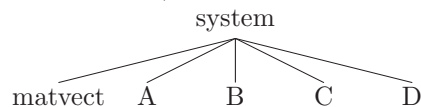
```

Le case réalise l'union des domaines de ses sous-expressions. Les deux points signifient la *restriction* de l'expression en partie droite au domaine en partie gauche. Le point applique la *fonction affine* en partie droite à son expression en partie gauche (il est le plus prioritaire des opérateurs). Les opérateurs arithmétiques ont une sémantique point-à-point (en particulier le * est une multiplication point-à-point).

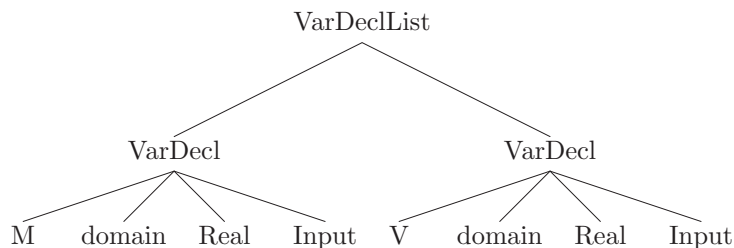
Question 10-1

Dessinez l'arbre syntaxique abstrait (AST) de ce programme.

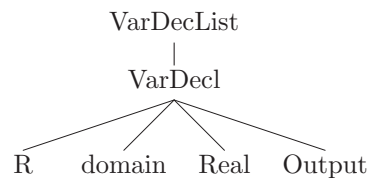
Voici l'AST du programme précédent (les domaines et expressions n'ont pas été détaillés) :



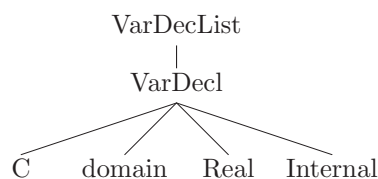
Avec l'arbre A :



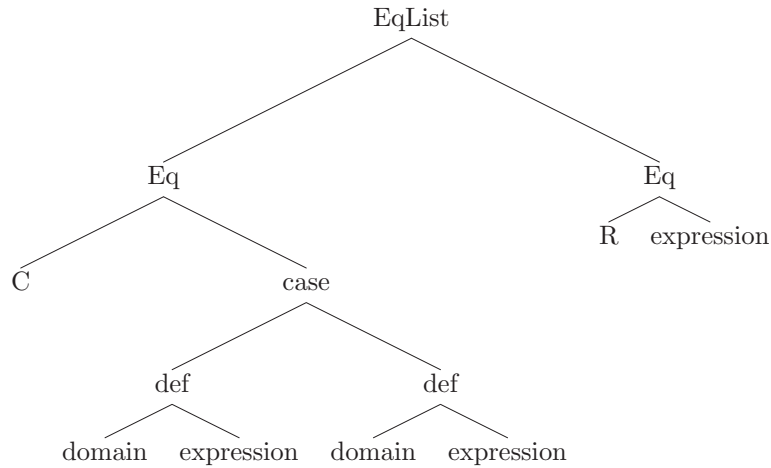
L'arbre B :



L'arbre C :



Enfin l'arbre D :



Question 10-2

Donnez la grammaire d'une fonction affine, la grammaire d'un domaine, la grammaire d'une expression ALPHA que l'on supposera toujours sous la forme *case-restriction-dépendance*, enfin la grammaire d'un programme.

System	→	<u>system</u> <u>id</u> (<u>VarDeclList</u>) <u>returns</u> (<u>VarDeclList</u>) <u>var</u> <u>VarDeclList</u> ; <u>let</u> <u>EquationList</u> <u>tel</u>
VarDeclList	→	<u>VarDeclList</u> ; <u>VarDecl</u>
VarDecl	→	<u>id</u> : <u>Domain</u> <u>of</u> <u>Type</u> ;
Type	→	<u>real</u> <u>bool</u> <u>integer</u>
Domain	→	{ <u>IdList</u> <u>IneqList</u> }
IdList	→	<u>IdList</u> , <u>id</u> <u>id</u>
IneqList	→	<u>IneqList</u> ; <u>Ineq</u> <u>Ineq</u>
Ineq	→	<u>AffineExp</u> <u>cmp</u> <u>AffineExp</u>
AffineExp	→	<u>TermSum</u> <u>-</u> <u>TermSum</u>
TermSum	→	<u>TermSum</u> <u>+</u> <u>Term</u> <u>TermSum</u> <u>-</u> <u>Term</u> <u>Term</u>
Term	→	<u>integer</u> <u>id</u> <u>id</u> <u>integer</u>
EquationList	→	<u>EquationList</u> ; <u>Equation</u> <u>Equation</u>
Equation	→	<u>id</u> <u>=</u> <u>Expression</u>
Expression	→	<u>Expression</u> <u>+</u> <u>Term</u> <u>Expression</u> <u>-</u> <u>Term</u> <u>Term</u> <u>-</u> <u>Term</u> <u>Domain</u> : <u>Expression</u>
Term	→	<u>Term</u> * <u>Factor</u> <u>Term</u> / <u>Factor</u> <u>Factor</u>

Factor	→	case ExpressionList esac
		Factor ; AffineFunction
		(Expression)
ExpressionList	→	ExpressionList ; Expression
		Expression
AffineFunction	→	(IdList $_ \geq$ AffineExpList)
AffineExpList	→	AffineExpList ; AffineExp
		AffineExpList

11 Domaine d'une expression

Voici les règles définissant le calcul du domaine d'une expression (qui proviennent de la sémantique du langage).

Constantes : $Dom(c) = \mathbb{Z}^0$

Variables : $Dom(V)$ est déclaré dans l'en-tête

Opérateurs unaires : $Dom(-e) = Dom(e)$

Opérateurs binaires : $Dom(e_1 \oplus e_2) = Dom(e_1) \cap Dom(e_2)$

Dépendance affine : $Dom(e.f) = f^{-1}(Dom(e))$

Opérateur case : $Dom(\text{case } e_1; \dots; e_n; \text{esac}) = \bigcup_{i=1}^n Dom(e_i)$

Question 11-1

Formalisez le calcul de l'attribut "domaine" dans votre grammaire. Quel type de grammaire à attributs obtenez-vous ?

Le traitement des constantes ne pose pas de problèmes. Le traitement des variables nécessite une table pour stocker leur domaine au moment de leur déclaration. Si on cherche à évaluer le domaine d'une variable au cours du traitement d'une expression, il suffit de récupérer la valeur du domaine dans la table.

VarDecl	→	id ; Domain of Type ;	: Add (Table, id, Domain)
Term	→	id	: Table (id)
		integer	: \mathbb{Z}^0

Le traitement des opérateurs binaires et unaires est direct.

Expression	→	Expression \pm Term	: $$$ \leftarrow \$1 \cap \$2$
		Expression $_$ Term	: $$$ \leftarrow \$1 \cap \$2$
		Term	: $$$ \leftarrow \1
		$_$ Term	: $$$ \leftarrow \1
		Domain ; Expression	: $$$ \leftarrow \1
Term	→	Term $_*$ Factor	: $$$ \leftarrow \$1 \cap \$2$
		Term $_ /$ Factor	: $$$ \leftarrow \$1 \cap \$2$
		Factor	: $$$ \leftarrow \1

Traiter le **case** nécessite de calculer le domaine d'une liste d'expressions :

ExpressionList	→	ExpressionList ; Expression	: $$$ \leftarrow \$1 \cap \$2$
		Expression	: $$$ \leftarrow \1
Factor	→	case ExpressionList esac	: $$$ \leftarrow \1

Enfin on suppose que l'attribut d'une fonction affine est la valeur de sa réciproque :

Factor	→	Factor ; AffineFunction	: $$$ \leftarrow \$2(\$1)$
---------------	---	--------------------------------	----------------------------

On obtient ainsi une grammaire à attributs synthétisés.

Question 11-2

Définissez précisément les cas d'erreur avec les messages correspondants : incompatibilité de dimension, surdéfinition de certaines valeurs, domaines vides... Essayez d'éviter les erreurs cascades.

Une équation de la forme **id** \equiv **Expression** donne la valeur de **id**. Il faut donc vérifier que **Expression** définit une valeur au moins pour tous les éléments du domaine de **id**, i.e. il s'agit de vérifier que le domaine de **id** est bien inclus dans le domaine de **Expression**. On doit avoir

notamment la dimension du domaine de `id` inférieure ou égale à celle de **Expression**. On peut également vérifier que les domaines considérés sont bien non vides (il est rare que le programmeur ait besoin d'une expression de domaine vide).

12 Forme compositionnelle et forme tableau

Pour vendre ALPHA au grand public, il a été mis au point une forme tableau qui est plus lisible. Voici le même programme sous forme tableau, jugez vous-même.

```
system matvect
    (M : {i,j | 1<=i; i<=42; 1<=j; j<=42} of real;
     V : {j | 1<=j; j<=42} of real)
    returns (R : {i | 1<=i; i<=42} of real);
var
    C : {i,j | 1<=i<=42; 0<=j<=42} of real;
let
    C[i,j] =
        case
            { | j=0 } : 0[];
            { | j>0 } : C[i,j-1] + M[i,j] * V[j];
        esac;
    R[i] = C[i,42];
tel;
```

Question 12-1

Proposez deux solutions différentes pour que l'analyseur syntaxique gère cette forme tableau ?

1. Avec attributs. Quel type d'attributs sont utilisés ?
2. Sans attributs.

Dans la forme tableau, les fonctions affines n'apparaissent plus explicitement ; l'analyseur syntaxique doit les retrouver. Ainsi, les variables `i`, `j` apparaissant dans le membre droit de l'équation `case ...` sont les mêmes que celles du membre gauche `C[i, j]` : cela doit être reconstruit au moment de l'analyse syntaxique. Cela peut être fait en utilisant des attributs hérités (le membre gauche hérite des variables du membre droit) ou une variable globale (qui stocke les variables de l'équation considérée).

13 Contexte d'une expression

Le *domaine de contexte*, ou plus simplement *contexte* d'une expression, est une notion qui a une définition intuitive simple : c'est le domaine sur lequel cette expression va servir à quelque chose. Cette notion dépend donc évidemment de l'environnement dans lequel on considère l'expression, d'où son nom.

Par exemple, si l'expression considérée est la partie droite d'une équation, son contexte est le domaine de la variable en partie gauche : en effet si l'expression possède un domaine plus grand que celui de cette variable en partie gauche, elle y est inutile.

A quoi sert cette notion de contexte ? Elle est utilisée principalement par toute transformation introduisant une *variable auxiliaire* pour remplacer une expression : une telle transformation doit déterminer automatiquement le domaine de cette nouvelle variable. Ce domaine est précisément le domaine de contexte de l'expression à remplacer : ainsi on ajoute le strict minimum de points de calculs au programme. L'introduction de variables auxiliaires est une transformation de programme fréquente.

Voici la liste des règles qui définissent le contexte $Cxt(e)$ d'une expression ALPHA e valide.

Racine : Si e est partie droite d'une équation $V = e$ alors $Cxt(e) = Dom(V)$. Sinon (expression flottante) alors $Cxt(e) = \mathbb{Z}^n$ où n est la dimension de l'expression.

Restriction : Si e apparaît dans le contexte $D : e$ alors $Cxt(e) = D \cap Cxt(D : e)$

Fonction affine : Si e apparaît dans le contexte $e.f$ alors $Cxt(e) = domImage(Cxt(e.f), f)$

Opérateurs unaires : Si e apparaît dans le contexte $-e$ alors $Cxt(e) = Cxt(-e)$

Opérateurs binaires : Si e apparaît dans le contexte $e' + e$ alors $Cxt(e) = Dom(e') \cap Cxt(e' + e)$

Question 13-1

Modifiez votre grammaire pour y ajouter l'attribut Contexte.

Le calcul du contexte d'une expression e située à un nœud (distinct de la racine) de l'arbre syntaxique ne peut se faire que si le contexte de l'expression e' englobant e a déjà été calculé : le contexte est donc un attribut hérité.

Equation	→	id \equiv Expression
Expression.Contexte	←	id.Domain
Expression1	→	Domain $_$ Expression2
Expression2.Contexte	←	Domain.Domaine \cap Expression1.Contexte
Factor1	→	Factor2 $_$ AffineFunction
Factor2.Contexte	←	domImage (Factor1.Contexte, f)
Expression	→	$_$ Term
Term.Contexte	←	Expression.Contexte
Expression1	→	Expression2 \pm Term
Term.Contexte	←	Expression2.Domaine \cap Expression1.Contexte

Le calcul de l'attribut Domaine se fait comme précédemment.

Correction du partiel 2003-2004

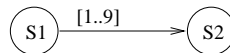
*La correction du partiel à été faite par Tanguy Risset et/ou Florent de Dinechin puis
partiellement complétée par Fabrice Rastello*

14 Expressions régulières

On considère l'expression régulière E_1 qui permet de décrire les entiers (on rappelle que la notation $[1..4]$ est un raccourci pour $1 \mid 2 \mid 3 \mid 4$) :

$$E_1 = [1..9]([0..9])^* \mid 0$$

Dans cette partie, on utilisera une généralisation commune de la définition d'un automate : on autorise les transitions étiquetées par un ensemble de symboles. Par exemple, l'automate suivant :



effectue une transition de $S1$ à $S2$ en lisant un des entiers compris (au sens large) entre 1 et 9.

Question 14-1

Rappeler brièvement comment, étant donné une expression régulière qui décrit un langage \mathcal{L} , on peut construire de façon systématique un automate fini non déterministe qui reconnaît le langage \mathcal{L} . Donner ensuite l'automate fini non déterministe obtenu par cette méthode pour l'expression régulière E_1 ci-dessus.

Contruction d'un automote non-déterminste à l'aide d'epsilon transitions, puis déterminisation de l'automate. A chaque étape, un seul état initial, un seul état final.

```

exp_0: si_0 -> sf_0
exp_1: si_1 -> sf_1
exp_0 | exp_1: si_01 -epsilon-> sf_01
               |_epsilon-~----|

```

... page 15 du cours

```

s0 -> s1-[1..9]->s2 -> s3->s4-[0..9]->s5->s6
|               | ^__epsilon__^
|               |_____|
|_____0_____

```

Question 14-2

Donner un automate fini déterministe qui décrit le même langage que l'expression régulière E_1 . Comment savoir si cet automate est minimal ou pas ?

déterminisation p15-16 du poly.

```

s1 -[1..9]-> s2f -
|           ^ | [0..9]
0           |__|

```

|
s3f

Minimal au sens de la procédure de minimisation vue en cours. Donc minimal!

Initialement, $s1$ et $\{s2, s3\}$. Puis $[0..9]$ séparent $s2$ et $s3$ ($s2 \rightarrow puit$ et $s3 \rightarrow s3$).

Question 14-3

On suppose disposer d'une primitive *nextChar()* qui renvoi le caractère présent sur l'entrée standard d'un programme. Écrivez en pseudo code un analyseur lexical pour reconnaître un entier d'après votre automate déterministe.

```
lastacceptant=-1;
si=s1;
while (si!=puit) {
    case si:
        s1: case nextChar:
        0: si=s3f; shift;
        _: si=puit;
        s2f: case nextChar:
            0..9: si=s2f; shift;
            _: si=puit;
        s3f: si=puit;
        if si in {s2f,s3f} lastacceptant=si;
}
```

15 Procédure

Considérons le programme C suivant :

```
void A(int val) {
    void C(int i) {
        void B() {
            printf("B appelé, val=%d",val) ;
        }
        if (i == 0) B() ; else C(i-1) ;
    }
    C(3) ;
}
```

Question 15-1

Dessiner la pile résultant de l'appel de A(5). (on dessinera la pile après chaque appel de procédure en indiquant ou pointent : le pointeur sur l'enregistrement d'activation (ARP), le lien d'accès statique et les valeurs connues).

Voici l'état de la pile (je n'ai pas représenté la sauvegarde des registres)

Question 16-2

Donner une grammaire G' équivalente qui est $LL(1)$. On ne demande pas de prouver formellement que les grammaires sont équivalentes, en revanche il faudra montrer précisément que votre grammaire est $LL(1)$ (en calculant les ensembles *First* et *Follow* en particulier).

1. dérécursivation de $Prog$: on ajoute un non-terminal $Prog'$.

$$P = \left\{ \begin{array}{l|l} \begin{array}{l} But \Rightarrow Prog\ eof \\ Prog \rightarrow I\ Prog' \\ Prog' \rightarrow ;\ I\ Prog' \\ \quad \quad \quad | \quad \epsilon \end{array} & \begin{array}{l} I \rightarrow pop() \\ \quad \quad | \quad pop(int) \\ \quad \quad | \quad push(num) \end{array} \end{array} \right\}$$

2. factorisation à gauche : pour $pop($ on ajoute un non-terminal RI ,

$$P = \left\{ \begin{array}{l|l} \begin{array}{l} But \Rightarrow Prog\ eof \\ Prog \rightarrow I\ Prog' \\ Prog' \rightarrow ;\ I\ Prog' \\ \quad \quad \quad | \quad \epsilon \end{array} & \begin{array}{l} I \rightarrow pop(\ RI \\ \quad \quad | \quad push(num) \\ RI \rightarrow) \\ \quad \quad | \quad int) \end{array} \end{array} \right\}$$

3. ensembles *First* : l'algorithme fixe les ensembles suivants :

$$First(I) = \{pop, push\}, \quad First(RI) = \{), int\}$$

$$First(Prog') = \{;, \epsilon\}$$

puis en itérant :

$$First(Prog) = \{pop, push\}$$

et les ensembles *First* ne bougent plus.

Comme certains ensembles comportent ϵ , on est obligé de calculer les ensembles *Follow*.

$But \rightarrow Prog\ eof$ implique $Follow(Prog) \leftarrow \{eof\}$. La règle $Prog \rightarrow I\ Prog'$ entraîne $Follow(Prog') \leftarrow Follow(Prog) = \{eof\}$, et $Follow(I) \leftarrow (First(Prog') - \epsilon) \cup Follow(Prog') = \{;, eof\}$. La règle $I \rightarrow pop(\ RI$ entraîne $Follow(RI) \leftarrow Follow(I) = \{;, eof\}$. D'où les valeurs :

$$Follow(Prog) = \{eof\} \quad Follow(Prog') = \{eof\}$$

$$Follow(RI) \leftarrow Follow(I) = \{;, eof\}$$

On peut donc définir donc $First^+(X) = First(X)$ si $\epsilon \notin X$ et $First(X) \cup Follow(X)$ sinon :

$$First^+(I) = \{pop, push\}, \quad First^+(RI) = \{), num\}$$

$$First^+(RProg) = \{;, eof\} \quad First^+(Prog') = \{;, eof\}$$

$$First^+(Prog) = \{pop, push\}$$

En fait on n'en a pas vraiment besoin, il suffit remarquer que le seule non-terminal qui peut poser problème pour un parsing $LL(1)$ est $Prog'$ car toutes les alternatives pour les autres non-terminaux commencent par des terminaux différents. Or on a :

$$First(;) \cap Follow(Prog') = \emptyset$$

Donc il n'y a pas de problèmes pour ces non terminaux non-plus, la grammaire est bien $LL(1)$.

Question 16-3

On revient maintenant à la grammaire G définie ci-dessus. construire l'automate $LR(1)$.

pour la grammaire LR(1) originale On part de la grammaire :

$$P = \left\{ \begin{array}{lcl} 1 & But & \rightarrow Prog \ eof \\ 2 & Prog & \rightarrow Prog ; I \\ 3 & & | I \end{array} \middle| \begin{array}{lcl} 4 & I & \rightarrow pop() \\ 5 & & | pop(int) \\ 6 & & | push(num) \end{array} \right\}$$

20 mn pour construire l'automate LR(1) , on part de

$$\begin{aligned} s_0 &= closure\left(\left\{ \begin{array}{l} [Prog \rightarrow \bullet Prog ; I, eof] \\ [Prog \rightarrow \bullet I, eof] \end{array} \right\}\right) \\ &= \left\{ \begin{array}{l} [Prog \rightarrow \bullet Prog ; I, eof] \\ [Prog \rightarrow \bullet I, eof] \\ [I \rightarrow \bullet pop(), eof] \\ [I \rightarrow \bullet pop(int), eof] \\ [I \rightarrow \bullet push(num), eof] \\ [Prog \rightarrow \bullet Prog ; I, ;] \\ [Prog \rightarrow \bullet I, ;] \\ [I \rightarrow \bullet pop(), ;] \\ [I \rightarrow \bullet pop(int), ;] \\ [I \rightarrow \bullet push(num), ;] \end{array} \right\} \\ s_1 &= Goto(s_0, Prog) = \left\{ \begin{array}{l} [Prog \rightarrow Prog \bullet ; I, eof] \\ [Prog \rightarrow Prog \bullet ; I, ;] \end{array} \right\} \\ s_2 &= Goto(s_1, ;) = \left\{ \begin{array}{l} [Prog \rightarrow Prog ; \bullet I, eof] \\ [Prog \rightarrow Prog ; \bullet I, ;] \\ [I \rightarrow \bullet pop(), eof] \\ [I \rightarrow \bullet pop(int), eof] \\ [I \rightarrow \bullet push(num), eof] \\ [I \rightarrow \bullet pop(), ;] \\ [I \rightarrow \bullet pop(int), ;] \\ [I \rightarrow \bullet push(num), ;] \end{array} \right\} \\ s_3 &= Goto(s_2, I) = \left\{ \begin{array}{l} [Prog \rightarrow Prog ; I \bullet , eof] \\ [Prog \rightarrow Prog ; I \bullet , ;] \end{array} \right\} \\ s_4 &= Goto(s_0, I) = \left\{ \begin{array}{l} [Prog \rightarrow I \bullet , eof] \\ [Prog \rightarrow I \bullet , ;] \end{array} \right\} \\ s_5 &= Goto(s_0, pop) = Goto(s_2, pop) = \left\{ \begin{array}{l} [I \rightarrow pop \bullet (), eof] \\ [I \rightarrow pop \bullet (int), eof] \\ [I \rightarrow pop \bullet (), ;] \\ [I \rightarrow pop \bullet (int), ;] \end{array} \right\} \\ s_6 &= Goto(s_5, ()) = \left\{ \begin{array}{l} [I \rightarrow pop (\bullet), eof] \\ [I \rightarrow pop (\bullet int), eof] \\ [I \rightarrow pop (\bullet), ;] \\ [I \rightarrow pop (\bullet int), ;] \end{array} \right\} \\ s_7 &= Goto(s_6,) = \left\{ \begin{array}{l} [I \rightarrow pop () \bullet , eof] \\ [I \rightarrow pop () \bullet , ;] \end{array} \right\} \\ s_8 &= Goto(s_6, int) = \left\{ \begin{array}{l} [I \rightarrow pop (int \bullet), eof] \\ [I \rightarrow pop (int \bullet), ;] \end{array} \right\} \\ s_9 &= Goto(s_8,) = \left\{ \begin{array}{l} [I \rightarrow pop (int) \bullet , eof] \\ [I \rightarrow pop (int) \bullet , ;] \end{array} \right\} \end{aligned}$$

$$s_{10} = Goto(s_0, push) = Goto(s_2, push) = \left\{ \begin{array}{l} [I \rightarrow push \bullet (num), eof] \\ [I \rightarrow push \bullet (num), ;] \end{array} \right\}$$

$$s_{11} = Goto(s_{10}, ()) = \left\{ \begin{array}{l} [I \rightarrow push(\bullet num), eof] \\ [I \rightarrow push(\bullet num), ;] \end{array} \right\}$$

$$s_{12} = Goto(s_{11}, num) = \left\{ \begin{array}{l} [I \rightarrow push(num \bullet), eof] \\ [I \rightarrow push(num \bullet), ;] \end{array} \right\}$$

$$s_{13} = Goto(s_{12}, ()) = \left\{ \begin{array}{l} [I \rightarrow push(num) \bullet, eof] \\ [I \rightarrow push(num) \bullet, ;] \end{array} \right\}$$

Il n'y a aucun conflit dans cet automate, donc la grammaire est $LR(1)$ pour être totalement clean il aurait fallu partir de la règle 1 :

$$s'_0 = closure(\{ [But \rightarrow \bullet Prog \ eof, \$] \})$$

$$= \left\{ \begin{array}{l} [But \rightarrow \bullet Prog \ eof, \$] \\ [Prog \rightarrow \bullet Prog \ ; \ I, eof] \\ [Prog \rightarrow \bullet I, eof] \\ [I \rightarrow \bullet pop(), eof] \\ [I \rightarrow \bullet pop(int), eof] \\ [I \rightarrow \bullet push(num), eof] \\ [Prog \rightarrow \bullet Prog \ ; \ I, ;] \\ [Prog \rightarrow \bullet I, ;] \\ [I \rightarrow \bullet pop(), ;] \\ [I \rightarrow \bullet pop(int), ;] \\ [I \rightarrow \bullet push(num), ;] \end{array} \right\}$$

$$s'_1 = Goto(s'_0, Prog) = \left\{ \begin{array}{l} [But \rightarrow Prog \bullet \ eof, \$] \\ [Prog \rightarrow Prog \bullet \ ; \ I, eof] \\ [Prog \rightarrow Prog \bullet \ ; \ I, ;] \end{array} \right\}$$

et $s'_f = Goto(s'_1, eof) = \{ [But \rightarrow Prog \ eof \ \bullet, \$] \}$ qui est l'état final.

Question 16-4

Pouvez-vous déduire de votre automate si cette grammaire G est $SLR(1)$? On rappelle que les états de l'automate $SLR(1)$ sont constitués d'ensembles d'item de la forme : $[A \rightarrow \alpha \bullet \beta, Follow(A)]$. On voit assez facilement que les *core* correspondant à deux lookahead symboles dans chaque état sont les mêmes. En d'autres termes, les états et les transitions de l'automate SLR vont être les mêmes que pour l'automate $LR(1)$, sauf que, par exemple l'état s_0 sera maintenant :

$$s_0 = \left\{ \begin{array}{l} [But \rightarrow \bullet Prog \ eof \ , \{\$ \}] \\ [Prog \rightarrow \bullet Prog \ ; \ I, \{;, eof\}] \\ [Prog \rightarrow \bullet I, \{;, eof\}] \\ [I \rightarrow \bullet pop(), \{;, eof\}] \\ [I \rightarrow \bullet pop(int), \{;, eof\}] \\ [I \rightarrow \bullet push(num), \{;, eof\}] \end{array} \right\}$$

On voit assez facilement sur l'automate que l'on a construit que cela n'introduit aucun conflit, ni en shift-reduce, ni en reduce-reduce. Donc la grammaire est bien $SLR(1)$.

Question 16-5

En vous aidant de l'algorithme d'un automate ascendant rappelé à la fin de ce sujet, décrire le comportement de l'automate lors de chacune des étapes de la reconnaissance de :

$pop(int); push(num); eof$

On pourra par exemple, représenter à chaque étape : la pile interne du parseur, l'état, l'action réalisée et le prochain caractère.

Voici les règles numérotées

$$P = \left\{ \begin{array}{l|l} \begin{array}{l} 1 \text{ } But \rightarrow Prog \text{ } eof \\ 2 \text{ } Prog \rightarrow Prog ; I \\ 3 \quad \quad | I \end{array} & \begin{array}{l} 4 \text{ } I \rightarrow pop() \\ 5 \quad \quad | pop(int) \\ 6 \quad \quad | push(num) \end{array} \end{array} \right\}$$

et voici la suite des actions pour le programme :

pop(int);
push(num); eof

<i>Etat</i>	<i>pile (av. ex.)</i>	<i>Entree (avant execution)</i>	<i>Action</i>
s'_0	<i>Invalid, s'_0</i> \perp	$\uparrow pop(int); push(num); eof$	<i>shift</i> $\rightarrow s_5$
s_5	<i>pop, s_5</i> <i>Invalid, s'_0</i> \perp	$pop \uparrow (int); push(num); eof$	<i>shift</i> $\rightarrow s_6$
s_6	$(, s_6$ <i>pop, s_5</i> <i>Invalid, s'_0</i> \perp	$pop(\uparrow int); push(num); eof$	<i>shift</i> $\rightarrow s_8$
s_8	<i>int, s_8</i> $(, s_6$ <i>pop, s_5</i> <i>Invalid, s'_0</i> \perp	$pop(int \uparrow); push(num); eof$	<i>shift</i> $\rightarrow s_9$
s_9	$), s_9$ <i>int, s_8</i> $(, s_6$ <i>pop, s_5</i> <i>Invalid, s'_0</i> \perp	$pop(int) \uparrow; push(num); eof$	<i>reduce</i> $5 \rightarrow s_4$
s_4	<i>I, s_4</i> <i>Invalid, s'_0</i> \perp	$pop(int) \uparrow; push(num); eof$	<i>reduce</i> $3 \rightarrow s'_1$

<i>Etat</i>	<i>pile</i>	<i>Entree</i>	<i>Action</i>
s'_1	$Prog, s'_1$ $Invalid, s'_0$ \perp	$pop(int) \uparrow; push(num); eof$	$shift \rightarrow s_2$
s_2	$;, s_2$ $Prog, s'_1$ $Invalid, s'_0$ \perp	$pop(int); \uparrow push(num); eof$	$shift \rightarrow s_{10}$
s_{10}	$push, s_{10}$ $;, s_2$ $Prog, s'_1$ $Invalid, s'_0$ \perp	$pop(int); push \uparrow (num); eof$	$shift \rightarrow s_{11}$
s_{11}	$(, s_{11}$ $push, s_{10}$ $;, s_2$ $Prog, s'_1$ $Invalid, s'_0$ \perp	$pop(int); push(\uparrow num); eof$	$shift \rightarrow s_{12}$
s_{12}	num, s_{12} $(, s_{11}$ $push, s_{10}$ $;, s_2$ $Prog, s'_1$ $Invalid, s'_0$ \perp	$pop(int); push(num \uparrow); eof$	$shift \rightarrow s_{13}$
s_{13}	$), s_{13}$ num, s_{12} $(, s_{11}$ $push, s_{10}$ $;, s_2$ $Prog, s'_1$ $Invalid, s'_0$ \perp	$pop(int); push(num) \uparrow; eof$	$reduce\ 6 \rightarrow s_3$
<i>Etat</i>	<i>pile</i>	<i>Entree</i>	<i>Action</i>
s_3	I, s_3 $;, s_2$ $Prog, s'_1$ $Invalid, s'_0$ \perp	$pop(int); push(num) \uparrow; eof$	$reduce\ 2 \rightarrow s'_1$
s'_1	$Prog, s'_1$ $Invalid, s'_0$ \perp	$pop(int); push(num) \uparrow; eof$	$shift \rightarrow s_2$
s_2	$;, s_2$ $Prog, s'_1$ $Invalid, s'_0$ \perp	$pop(int); push(num); \uparrow eof$	<i>ERREUR</i>

Question 16-6

On donne maintenant la sémantique suivante aux instructions (SP désigne le registre pointant sur le sommet de la pile, la valeur pointée par SP est à l'adresse $val(SP)$) :

- $push(num)$ empile l'entier num sur la pile. On suppose disposer d'une primitive $valeur(num)$ qui renvoie la valeur de l'entier. $push(num)$ range $valeur(num)$ à l'adresse $val(SP)$ puis décrémente SP de 4 (la pile grandit vers les petites adresses).
- $pop()$ dépile un entier, c'est à dire qu'on range la valeur pointée par SP dans r_0 , puis incrémente SP de 4.
- $pop(int)$ dépile $valeur(int)$ entiers : on range la valeur pointée par SP dans r_0 , la valeur de l'adresse $(val(SP) + 4)$ dans le registre r_1 , la valeur de l'adresse $(val(SP) + 8)$ dans le registre r_2 , etc. jusqu'à la valeur de l'adresse $(val(SP) + valeur(int) \times 4)$ dans le registre $r_{valeur(int)}$. Ensuite on incrémente SP de $valeur(int) \times 4$

Question : Concevoir un parseur "ad-hoc dirigé par la syntaxe" en utilisant la syntaxe "à la yacc", qui réalise les actions suivantes :

- reconnaît une phrase du langage décrit par G .
- produit, lors du parsing, le code Iloc correspondant aux opérations sur la pile spécifiées par le programme. Pour cela on dispose d'une primitive $Emit(op, oper1, oper2, res)$ qui produit l'opération Iloc $op \ oper1, oper2 \Rightarrow res$.
- calcule, lors du parsing, le nombre de mots empilées (c'est à dire, la différence entre la valeur de SP avant l'exécution du code et après).

Pour répondre, on se contentera de spécifier les actions associées aux règles en utilisant le système de nommage de yacc : \$\$ pour la partie gauche de la règle, \$1 pour le premier symbole en partie droite etc.

10 mn Voici un exemple d'actions par règles :

$$P = \left\{ \begin{array}{ll} 1 \text{ } But & \Rightarrow \text{ } Prog \text{ } eof & \begin{array}{l} \$\$ \leftarrow \$1 \end{array} \\ 2 \text{ } Prog & \rightarrow \text{ } Prog \text{ } ; \text{ } I & \begin{array}{l} \$\$ \leftarrow \$1 + \$3 \end{array} \\ 3 \text{ } Prog & \rightarrow \text{ } I & \begin{array}{l} \$\$ \leftarrow \$1 \end{array} \\ 4 \text{ } I & \rightarrow \text{ } pop(\text{ }) & \begin{array}{l} \$\$ \leftarrow -1 \\ Emit(load, sp, r_0) \\ Emit(addI, sp, 4, sp) \end{array} \\ 5 \text{ } I & \rightarrow \text{ } pop(\text{ } int \text{ }) & \begin{array}{l} \$\$ \leftarrow -valeur(int) \\ Do \text{ } i = 0 \text{ } to \text{ } valeur(int) \\ \quad Emit(loadAI, sp, 4 \times i, r_i) \\ \quad Emit(addI, sp, 4 \times valeur(int), sp) \end{array} \\ 6 \text{ } I & \rightarrow \text{ } push(\text{ } num \text{ }) & \begin{array}{l} s\$\$ \leftarrow 1 \\ Emit(load, valeur(\$3), r_{-1}) \\ Emit(store, r_{-1}, sp) \\ Emit(subI, sp, 4, sp) \end{array} \end{array} \right\}$$

17 Procédure en paramètre

Question 17-1

Certain langages impératifs offrent la possibilité de passer une procédure en paramètre. Proposer un mécanisme de génération de code, basé sur la gestion de pile classique, pour autoriser le passage de procédure en paramètre. Détaillez autant que possible votre description.

Le principe est de passer un descripteur de procédure avec un pointeur sur le code et un pointeur sur le lien static (cf le grune et bal).

18 Annexes

Voici l'algorithme du parseur $LR(1)$ à base de table, la syntaxe des instructions Iloc est disponible sur demande.

```
push Invalid
push  $s_0$ 
motCourant  $\leftarrow$  prochainMot()
tant que (True)
     $s \leftarrow$  sommet de pile
    si Action[ $s, motCourant$ ] = "shift  $s_i$ "
    alors push motCourant
        push  $s_i$ 
        motCourant  $\leftarrow$  prochainMot()
    sinon si Action[ $s, motCourant$ ] = "reduce  $A \rightarrow \beta$ "
    alors pop  $2 \times |\beta|$  symboles
         $s \leftarrow$  sommet de pile
        push  $A$ 
        push Goto[ $s, A$ ]
    sinon si Action[ $s, motCourant$ ] = "accept" alors Accept
    sinon Erreur
```

Optimisations de compilation

La correction de ce TD à été faite par Miriam Krüger et Alexis Ballier à partir des implémentations de Fabrice Rastello.

19 Allocation de registres sur un basic-bloc

Soit le code assembleur suivant :

```

Load b → R1
Load b → R2
Mult R1 R2 → R2
Make 4 → R3
Load a → R4
Load c → R5
Mult R4 R5 → R5
Mult R5 R3 → R3
Sub R2 R3 → R3
  
```

Question 19-1

Que calcule ce code et combien utilise-t-il de registres ?

L'état des registres à la fin de l'exécution du code est :

Registre	Contenu
R1	b
R2	b^2
R3	$b^2 - 4ac$
R4	a
R5	ac

On peut donc affirmer qu'il calcule $b^2 - 4ac$ et utilise 5 registres.

Question 19-2

Ecrire ce code en SSA. SSA signifie Static Single Assignment : on n'a pas le droit d'assigner *textuellement* deux fois une valeur à une variable. A l'opposé du SA utilisé en parallélisation automatique, qui implicitement correspond à de l'assignation unique dynamique c'est à dire durant l'exécution du programme.

Par exemple :

devient :
$$\text{if } (cond) \text{ then } x = \dots \text{ else } x = \dots \text{ fi}$$

$$\text{if } (cond) \text{ then } x_1 = \dots \text{ else } x_2 = \dots \text{ fi } x = \Phi(x_1, x_2)$$

Le code SSA est :

```

Load b → V1
Load b → V2
Mult V1 V2 → V3
Make 4 → V4
Load a → V5
Load c → V6
Mult V5 V6 → V7
Mult V4 V7 → V8
Sub V3 V8 → V9

```

Question 19-3

Effectuer l'allocation des registres à ordonancement fixé avec un nombre minimum de registres.

Le tableau suivant représente la durée de vie (live range) des registres au cours de l'exécution du code, avec la convention qu'un registre meurt juste avant sa dernière utilisation :

Code	R1	R2	R3	R4	R5	R6	R7	R8	R9
Load b → V1									
Load b → V2									
Mult V1 V2 → V3									
Make 4 → V4									
Load a → V5									
Load c → V6									
Mult V5 V6 → V7									
Mult V4 V7 → V8									
Sub V3 V8 → V9									

On voit qu'il y en a 4 utilisés en même temps, donc on est obligés d'utiliser au moins 4 registres.

L'algorithme glouton qui parcourt du haut vers le bas et qui alloue le code au premier registre libre est optimal. On peut remarquer que le graphe d'interférence (un sommet par variable, une arrête entre deux variable qui sont en vie en même temps à un endroit du code) d'un code linéaire est un graphe d'intervalle. Allouer des registres, c'est colorier le graphe. Le problème de coloriage sur un graphe d'intervalle est trivialement polynomial grâce à cet algorithme glouton.

Une fois l'allocation des registres effectuée, on obtient le code suivant :

```

Load b → R1
Load b → R2
Mult R1 R2 → R1
Make 4 → R2
Load a → R3
Load c → R4
Mult R3 R4 → R3
Mult R2 R4 → R4
Sub R1 R4 → R2

```

20 Génération de code et minimisation du nombre de registres sur un arbre d'expression

Question 20-1

Ecrire l'AST de cette expression : $(b^2 - 4ac)$.



Question 20-2

Peut-on effectuer le calcul sur 2 registres ?

Oui, la preuve :

```

Make 4 → R0
Load a → R1
Mult R0 R1 → R0
Load c → R1
Mult R0 R1 → R0
Load b → R1
Mult R1 R1 → R1
Sub R1 R0 → R0

```

Question 20-3

Générer le code à pile de l'expression.

```

Push 4
Push a
Mult
Push c
Mult
Push b
Push b
Mult
Sub em (qui va bien)

```

Question 20-4

Donner un algorithme polynomial qui génère le code à pile d'un arbre d'expression avec une hauteur de pile minimum.

On ne considère que les arbres binaires du type :

$$\begin{array}{c} \text{opération} \\ \swarrow \quad \searrow \\ A \quad B \end{array}$$

Lorsque l'on est dans ce cas, on doit choisir entre "exécuter A en premier, stocker son résultat puis exécuter B", ou bien "exécuter B en premier, stocker son résultat puis exécuter A".

Ainsi, selon le choix fait on a :

- Si on exécute A en premier, on a comme hauteur de pile : $\max(\text{hauteur}(A), 1 + \text{hauteur}(B))$.
- Si on exécute B en premier, on a comme hauteur de pile : $\max(\text{hauteur}(B), 1 + \text{hauteur}(A))$.

Ainsi, pour minimiser la hauteur de pile totale de l'arbre, si l'on suppose que A et B sont de hauteur de pile minimale il suffit d'exécuter le sous arbre A ou B qui a la hauteur de pile la plus grande en premier.

On obtient ainsi l'algorithme suivant :

```

let rec code = function
  Vide -> (0, Rien)
  | Arbre(operateur,A,B) -> let (i, code_de_A) = code(A) in
    let (j, code_de_B) = code(B) in
    if i > j then (i, code_de_A + operateur + code_de_B)

```

```
else (max(i+1, j), code_de_B + operateur + code_de_A);;
```

Minimiser le nombre de registres sur un DAG n'a pas de sens car un DAG ne peut pas nécessairement être implémenté par code à pile sans dupliquer les calculs.

Pour un code général (CFG) avec une architecture à 3 adresses, le problème est NP-Complet. [Chaitin, Chandra, Auslander : Register Allocation Via Graph Coloring, Computer Languages, 1981]

21 Minimisation de la longueur du code pour du code à un registre et deux adresse

Un code à un registre et deux adresse a :

- Un seul registre
- Trois types d'instructions : load, store, op

Par exemple, le calcul de $A \leftarrow B + C$ s'écrit :

```
load B
add C
store A
```

Le calcul de $B * C + D * E$ s'écrit :

```
load D
mult E
store T
load B
mult C
sub T
```

Question 21-1

Ecrire le code à deux adresses de longueur minimum de $b^2 - 4ac$.

Contrairement à la question précédente, quel que soit l'ordre dans lequel on développe le calcul, le coût est le même :

```
load a
mult c
mult 4
store T
load b
mult b
sub T
```

Par contre pour un graphe DAG quelconque, minimiser le nombre d'instructions est NP-Complet [Bruno, Sethi : Code generation for a one-register machine, Journal of the ACM, July 1976.]

22 Sélection d'instruction

On a un ensemble d'instructions de base, mettons $\Sigma = \{b_i, 0 \leq i \leq N\}$ et un sous ensemble \mathcal{I} tel que $\{\{b_i\}, 0 \leq i \leq N\} \subset \mathcal{I} \subset \{(b_{i_1}, \dots, b_{i_k}), k \in \mathbb{N}, 0 \leq b_{i_k} \leq N\}$ représentant les instructions disponibles sur notre processeur.

Ca veut dire que notre processeur sait au moins faire les opérations de base mais qu'il peut aussi faire des opérations effectuant plusieurs opérations de base d'un seul coup.

Exemple, sur un MMX, l'instruction : MAC A,B,C exécute : $A += B * C$.

Le but du jeu est de minimiser le nombre total d'instructions utilisées à partir d'un code écrit avec les instructions de base.

Question 22-1

Connaissant Σ et \mathcal{I} , et en ne changeant pas l'ordre des instructions, écrire un algorithme qui minimise le nombre d'instructions total à partir d'un code initial nommé \mathcal{C} .

L'idée de l'algorithme est la suivante : on note $\mathcal{C} = g\mathcal{C}'$ la succession de l'instruction $g \in \mathcal{I}$ et du reste du code \mathcal{C}' . Dans ce cas, on a

$$\text{opt}(\mathcal{C}) = \min_{g \in \mathcal{I}, \text{ tq } \mathcal{C} = g\mathcal{C}'} 1 + \text{opt}(\mathcal{C}')$$

On sait que l'optimum pour un code vide est un code vide, on peut ainsi appeler cet algorithme récursivement, ce qui donnerait un algorithme de complexité exponentielle.

On peut aussi trivialement construire un algorithme de type programmation dynamique. On utilise pour ça les notations suivantes :

- $\text{opt}(k) = \text{opt}(\mathcal{C}_k \mathcal{C}_{k-1} \dots \mathcal{C}_1)$ avec : si \mathcal{C} contient n instructions de base $\mathcal{C} = \mathcal{C}_n \mathcal{C}_{n-1} \dots \mathcal{C}_1$ (nummémentés en sens inverse).
- Si $\mathcal{C}' = \mathcal{C}_k \mathcal{C}_{k-1} \dots \mathcal{C}_1$, $\mathcal{C}' = g\mathcal{C}''$ avec $\text{opt}(\mathcal{C}') = 1 + \text{opt}(\mathcal{C}'')$ alors on pose $g_{\text{opt}}(k) = g$ (on en choisit un au hasard s'il y a plusieurs solutions).

L'algorithme remplit d'abord les tableaux opt et g_{opt} :

```

opt(0) = 0
g_opt(0) = ∅
for k = 1 to n
    opt(k) = k
    forall g ∈ I tq g = C_k C_{k-1} ... C_{k-|g|+1}
        if 1 + opt(k - |g|) < opt(k) then
            opt(k) = 1 + opt(k - |g|)
            g_opt(k) = g

```

ca sert à rien c'est juste pour faire joli

une borne sup

cet ensemble est non vide

Ensuite, on extrait une solution optimale : $k = n$

```

while k > 0
    print g_opt(k)
    k = k - |g_opt|

```

Considérons par exemple le code à 9 instructions représenté par la chaîne suivante :



Les lignes de couleur représentent les groupements possibles : $\mathcal{I} = \{a, b, c, d, ab, ba, bbcd\}$.

En faisant tourner l'algorithme, on obtient :

programme	a	b	a	b	b	c	d	a	b
opt	4	3	3	2	4	3	2	1	1
g_{opt}	a	ba	a	bbcd	b	c	d	ab	b

Ce qui donne le regroupement suivant à 4 instructions : $(a)(ba)(bbcd)(ab)$. On remarquera, aurait pu valoir ab , ce qui aurait donné comme solution équivalente : $(ab)(a)(bbcd)(ab)$.

Question 22-2

On vient de faire un algorithme de groupement de code optimal pour un code linéaire ; Peut-on le faire sur un arbre ?

Oui, on peut le faire, avec le même algorithme, en regroupant les instructions en sous-arbres à partir des feuilles de l'arbre (au lieu de regrouper en sous-chaînes à partir de la fin du programme comme précédemment). C'est ce qui est fait dans la dernière phase de l'algorithme dit *BURST*. La première phase permettant de trouver l'ensemble des groupements possibles à partir d'une description de l'architecture (ISA).

Question 22-3

Et sur un DAG ?

C'est NP-Complet. Une manière simple de le prouver est de faire une réduction à X3C (EXACT-COVER BY 3-SETS. Voir par exemple le "Garey & Johnson") :

1. X3C : un ensemble X avec $|X| = 3q$ et une collection C de parties de X à 3-éléments. Peut on partitionner X en q parties (disjointes) tq chacune de ces q parties soit dans C .
2. Pour une instance de X3C on considère un graphe biparti avec les éléments de X d'un coté, et les éléments C de l'autre, 3 arêtes partant de chaque C -sommets. A chaque sommet du graphe est associé une instruction, et pour chaque sommet de C $c = \{x, y, z\}$, on rajoute l'instruction qui regroupe le calcul de " $c(x(\dots), y(\dots), z(dots))$ ".
3. C' , sous ensemble de C , est un ensemble de regroupements. Il est valide, si pour tout $c_1 \neq c_2$ de C' , $c_1 \cap c_2 = \emptyset$. Le nombre d'instructions du code généré est $|X| + |C| - 3|C'|$. Plus C' est grand, meilleur le code est. On a $|C'| \leq q$, et si $|C'| = q$ alors C' est une solution à l'instance initiale de X3C.

Élimination de copies redondantes

La correction de ce TD à été faite par Laurent Jouhet. La correction n'a pas été vérifiée.

23 Copy propagation

23.1 Sur un Basic-bloc

La copie propagation est une transformation qui, étant donné une assignation du type `move x, y`², remplace les utilisations futures de x en y , des lors qu'aucune instruction n'a modifié ni la valeur de x ni celle de y . Par exemples :

Code initial	<code>move x, y</code>	<code>move x, y</code>	<code>move x, y</code>
		<code>add x, y, 2</code>	<code>add y, y, 1</code>
	<code>add z, x, 2</code>	<code>add z, x, 2</code>	<code>add z, x, 2</code>
Après copy-propagation	<code>move x, y</code>	inchangé	inchangé
	<code>add z, y, 2</code>		

Sur un basic-bloc (BB), l'algorithme est :

```

ACP = ∅ {ensemble de var × var}
for i = 1 to n do
  {remplace les operandes qui sont des copies}
  for all operand instri.usej do
    let b = instri.usej
    if ∃a st b × a ∈ ACP then
      instri.usej ← a
    end if
  {Efface de l'ACP les paires invalidees par l'assignation courante.}
  ...
  {Insere la paire dans l'ACP}
  if instri = move a, b then
    ACP = ACP ∪ {a × b}
  end if
end for
end for

```

Dans ce pseudo-code :

- les n instructions du basic-bloc sont indexées dans l'ordre $instr_1, instr_2, \dots, instr_n$,
- pour l'instruction $instr_i$, la j -ième opérande use (resp. def) est dénotée $instr_i.use_j$ (resp. $instr_i.def_j$).
- l'ensemble ACP implémente une table de hashage où si $a \times b \in ACP$, a est la clé, b la valeur.

²`move x, y` a la sémantique $x := y$

Question 23-1

Compléter l'algorithme (partie “*efface de l'ACP les paires invalidées...*”).

```

ACP = ∅ {ensemble de var × var}
for i = 1 to n do
    {remplace les operandes qui sont des copies}
    for all operand instri.usej do
        let b = instri.usej
        if ∃a st b × a ∈ ACP then
            instri.usej ← a
        end if
    end for
    {Efface de l'ACP les paires invalidees par l'assignation courante.}
    let X = instri.defj
    if ∃a st X × a ∈ ACP then
        remove X × a ∈ ACP
    end if
    if ∃a st a × X ∈ ACP then
        remove a × X ∈ ACP
    end if
    {Insere la paire dans l'ACP}
    if instri = move a, b then
        ACP = ACP ∪ {a × b}
    end if
end for

```

Question 23-2

Appliquer l'algorithme au code “UNIB” suivant :

UNIB :

```

S1 :   move   b, a
S2 :   add    c, b, 1
S3 :   move   d, b
S4 :   add    b, d, c
S5 :   move   b, d

```

Les différents webs (cf. 24.1) :

UNIB :	a	b			
S ₁ :	move	b, a	a ₁	b ₁	
S ₂ :	add	c, b, 1		b ₁	c ₁
S ₃ :	move	d, b		b ₁	c ₁ d ₁
S ₄ :	add	b, d, c	b ₂	c ₁	d ₁
S ₅ :	move	b, d	b ₃		d ₁

Après copy-propagation :

UNIB :	instruction	opérandes	ACP final
S ₁ :	move	b, a	(b, a)
S ₂ :	add	c, a, 1	(b, a)
S ₃ :	move	d, a	(b, a), (d, a)
S ₄ :	add	b, a, c	(d, a)
S ₅ :	move	b, a	(d, a), (b, a)

23.2 Sur un CFG

On définit par analyse data-flot les ensembles :

1. $COPY(BB_i)$: ensemble des instructions move du BB_i qui atteignent le bas du bloc : c'est à dire s'il n'existe pas d'instruction entre ce move et la fin du bloc qui l'invalidé (on dit aussi le tue). Par exemple, pour le code linéaire précédant, on aurait $COPY(UNIB) = \{S_5\}$ car S_1 est invalidé par S_4 et S_3 est invalidé par S_4 .

2. $KILL(BB_i)$: ensemble des copies du programme qui si elles étaient valides à l'entrée du bloc BB_i seraient tuées par l'une des instructions du bloc BB_i .

$$3. CPin(BB_i) = \bigcap_{BB_j \in pred BB_i} CPout(BB_j)$$

$$4. CPout(BB_i) = COPY(BB_i) \cup (CPin(BB_i) - KILL(BB_i))$$

Ensuite, on utilise $CPin$ pour initialiser ACP dans l'algorithme de copy-propagation local, que l'on tourne sur chaque BB.

Question 23-3

Les ensembles $CPin$ et $CPout$ sont calculés par méthode de point fixe. A quelle valeur doivent-ils être initialisés. Pourquoi ? En quelle mesure est-ce correct ?

On initialise le $CPin$ à \emptyset , et les $CPout(BB_i)$ avec $COPY(BB_i)$. De plus, il faut initialiser $CPout(entry)$ avec toutes les instructions *move*. En effet, le but de cet algorithme est de garder un maximum d'instructions *move* d'un bloc à l'autre, et de supprimer seulement les *move* qui vont être effacés par un autre. Ainsi, on ne s'occupe pas de savoir si une variable est initialisée ou non, mais seulement si une instruction *move* va l'effacer. L'initialisation permet de n'effacer aucun *move* sans raison ; $CPout$ permet d'effacer tous les *move* nécessaires dans les blocs fils ; L'initialisation de $CPout(empty)$ permet de garder toutes les instructions *move* même si les variables ne sont pas initialisées.

Question 23-4

Appliquez l'algorithme de copy-propagation global sur le code de la Figure 5.

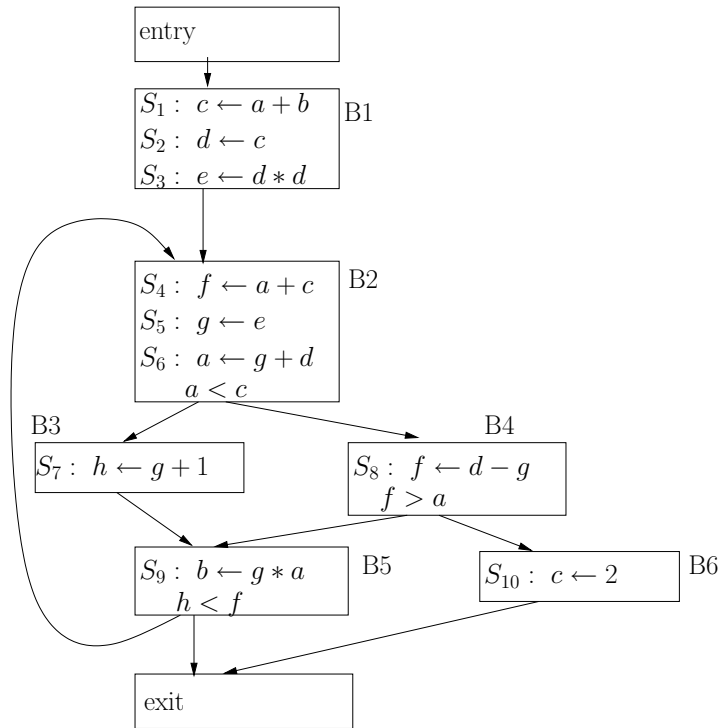


FIG. 5 – Un graphe de flot de controle (CFG).

	COPY	KILL	CPout	CPin
entry	/	/	S_2, S_5	/
B1	S_2	S_2, S_5	S_2, S_5	S_2, S_5
			S_2	\emptyset
B2	S_5	\emptyset	S_2, S_5	S_2, S_5
				S_2
B3	\emptyset	\emptyset	S_2, S_5	S_2, S_5
B4	\emptyset	\emptyset	S_2, S_5	S_2, S_5
B5	\emptyset	\emptyset	S_2, S_5	S_2, S_5
B6	\emptyset	S_2	S_2, S_5	S_2, S_5
			S_5	
exit	/	/	/	S_2, S_5
				S_5

Question 23-5

Le résultat serait-il identique si l'on tournait en plus l'algorithme de copy-propagation local sur chacun des blocs avant l'algorithme de propagation global ?

On peut améliorer l'algorithme de point fixe en faisant tourner l'algorithme de propagation locale sur chaque bloc : en effet, l'algorithme de point fixe manipule des copies des variables, alors que la propagation locale manipule les valeurs des variables. Ainsi on peut propager les valeurs avant d'utiliser les copies pour plus d'efficacité.

Question 23-6

Donnez plusieurs exemples où l'algorithme est incapable d'enlever certaines copies pourtant simplifiables.

Lorsque Une variable est définie différemment dans deux blocs mais prend toujours la même valeur : si on a un bloc `'move a, b'` qui a deux prédcesseurs `'move b, 2'` et `'move c, 2 ; move b, c'`, lors du calcul de *CPin* et *CPout*, ces valeurs seront retirées alors que l'exécution de copy propagation de manière plus locale les aurait repérées comme étant égales.

Plus simple : Deux blocs (ayant des indices différents) effectuant la même instruction *move* devraient être simplifiés, mais ils ne le seront pas grâce à l'algorithme de point fixe, car il travaille sur les copies des variables, qui dans ce cas sont différentes.

24 variable coalescing

24.1 web

Une même variable peut être utilisée par une même routine pour des utilisations différentes non corrélées (e.g. la variable *i* pour chaque boucle). Un web sépare les instances séparables d'une même variable.

Par exemple :

```
x:=2
y:=4
w:=x+y
z:=x+1
u=x*y
x=z*2
```

Les deux assignations à *x* sont décorrélées. Formellement, les webs représentent les parties connexes des chaînes *def-use* d'une même variable ou d'un même registre.

Question 24-1

Donner les webs de la Figure 6.

Un web pour une variable est une union maximale de chaînes définition-utilisation relatives à cette variable, et intersectant 2 à 2.

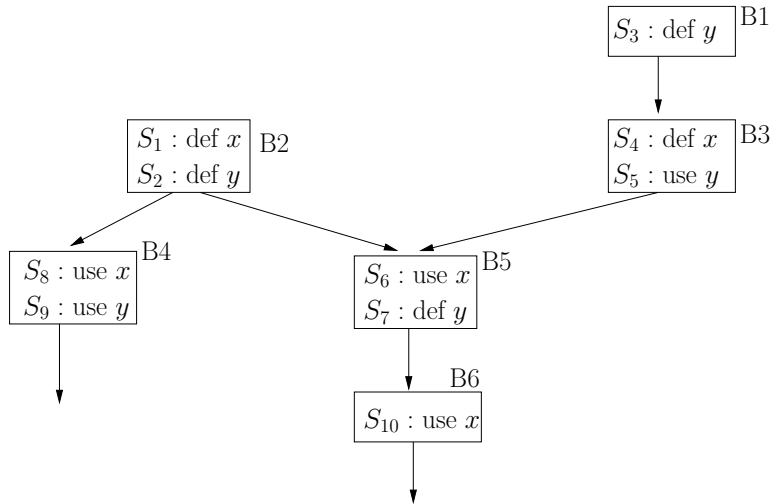


FIG. 6 – Un autre graphe de flot de controle (CFG).

On a un web par valeur du programme : Pour x : $x_1 : (S_1, S_4, S_6, S_8, S_{10})$. Pour y : $y_1 : (S_2, S_9)$, $y_2 : (S_7)$, $y_3 : (S_3, S_5)$.

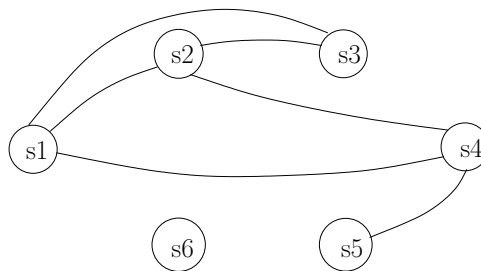
24.2 Graphe d'interference

	s1	s2	s3	s4	s5	s6
s1 :=2	X					
s2 :=4	X	X				
Soit le code s3 :=s1+s2	X	X	X			
s4 :=s1+1	X	X		X		
s5 :=s1*s2	X	X		X	X	
s6 :=s4*2				X		X

Question 24-2

Donner le graphe d'interférence de ce code.

Deux variables interfèrent lorsque elles sont vivantes en même temps.



Question 24-3

Donner le graphe d'interférence de la Figure 6.

24.3 Coalescing

L'algorithme de register coalescing consiste à partir du graphe d'interférence, d'y rajouter des pointillés entre chaque variable reliée par une instruction de copie (affinité), et de fusionner un

maximum de tels couples reliés par un pointillé.

Question 24-4

Donner le code correspondant au graphe suivant :



Ce graphe d'affinité correspond à

- move a, b ; (affinité)
- add c, a, b ; (interférence)

Question 24-5

Construire le graphe d'interférence et d'affinité du code UNIB en supposant que $LIVEin(UNIB) = \{a\}$ et $LIVEout(UNIB) = \{b\}$. Effectuer le coalescing dessus.

cf. figure 7

Question 24-6

Même question, en supposant que $LIVEout(UNIB) = \{a, b\}$. Peut-on modifier la notion d'interférence classique ?

cf. figure 8

a est en vie tout le long du programme et interfère avec toutes les autres variables :

Modification de la notion classique : ...

Question 24-7

Trouvez un exemple pour lequel, en fonction de l'ordre dans lequel on coalesce les variables, on obtient un résultat meilleur qu'un autre.

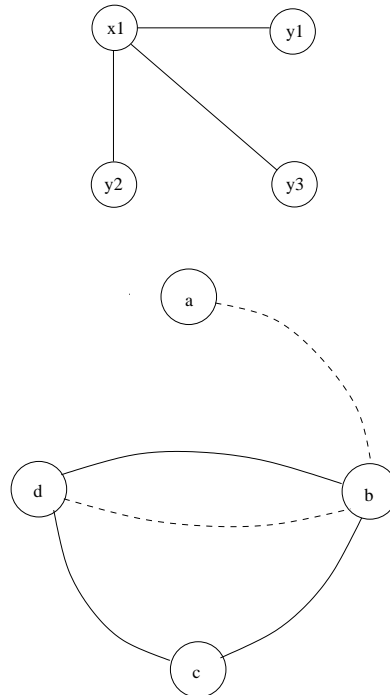


FIG. 7 – affinité de UNIB pour LIVEout = b

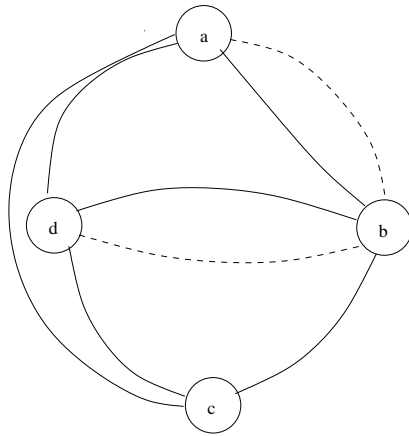


FIG. 8 – affinité de UNIB pour LIVEout = a, b

Static Single Assignment

Pas de correction pour ce TD...

On peut utiliser la constante `undef`.

On note Li le label d'entête pour un basic-bloc donné. Un basic-bloc étant une succession d'instructions du programme avec pour seule instruction `jump` sa dernière instruction et pour seul label sa première instruction.

25 Propriétés de la forme SSA

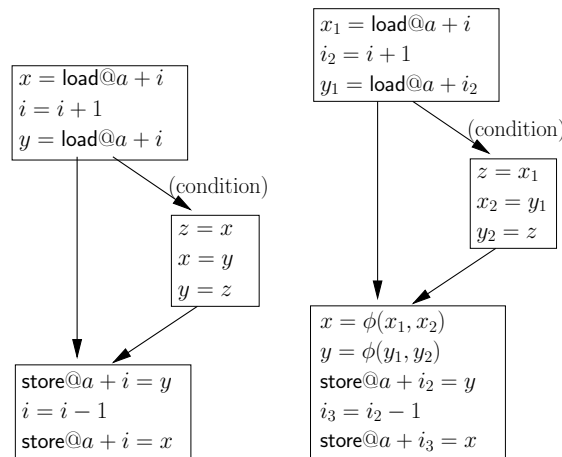


FIG. 9 – Forme initiale et forme SSA correspondante

25.1 Exemples pour faire manipuler SSA

Question 25-1 Donner la forme SSA pour les variables x & y du code suivant :

```

x = 0
b = false
while une condition do
    if une autre condition then
        b = true
        x++
        y = 1/x
    end if
end while

```

```

if  $b$  then
     $a[x] = y$ 
end if

```

Question 25-2 La copy-propagation est une passe qui élimine les copies (instruction move). Le principe sous SSA est le suivant : à chaque fois que l'on a une copie (b vers a), on élimine cette copie en remplaçant textuellement "a" partout où il apparaît par "b". Par exemple :

$b = \dots$	donne	$b = \dots$
$c = \dots$		$c = \dots$
$a = b$		if <i>condition</i> then
if <i>condition</i> then		...
...		else
else		...
...		end if
end if		$d = \phi(b, c)$
$d = \phi(a, c)$		$\dots = d$
$\dots = d$		

Appliquez cette transformation à l'exemple de la Figure 9.

25.2 Dominance

On dit qu'un bloc $L1$ domine un bloc $L2$, si tout chemin du CFG de la racine de la procédure à $L2$ passe par $L1$.

Question 25-3 En pratique, dynamiquement (à l'exécution) si toute exécution de $L2$ est précédée par une exécution de $L1$, $L1$ ne domine pas nécessairement $L2$: l'ensemble des chemins possibles pour la notion de dominance ne prennent pas en compte les possibles relations qui existent entre les différentes conditions de branchement. Construire un exemple qui illustre ce point.

Ensuite, $L1$ est un dominateur immédiat de $L2$ s'il n'existe pas de bloc $L3$ différent de $L1$ et $L2$ dominé par $L1$ et dominant $L2$.

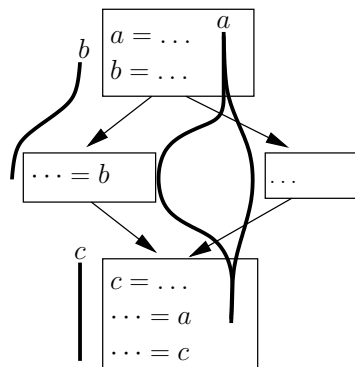
On considère le graphe de dominance $G = (V, E)$ où les sommets V sont les basic-blocs du graphe de flot de contrôle (CFG) et $(A, B) \in E$ si et seulement si A est un dominateur immédiat de B .

Question 25-4 Montrer que G est un arbre de racine le bloc entry.

L'unique dominateur immédiat d'un basic-bloc ou la liste des dominés font partie des données pré-calculées mises à la disposition des phases d'optimisation qui travaillent sur le graphe de flot de contrôle.

25.3 Vie d'une variable. live range

On appelle durée de vie d'une variable, l'ensemble des points du CFG où cette variable doit être maintenue en vie : dans un basic-bloc cela correspond à un intervalle entre sa définition et sa dernière utilisation. Par exemple,



La définition initiale d'interférence était la suivante : "deux variables interfèrent si et seulement si leur live-range s'intersectent". Cette définition a ensuite été améliorée en : "deux variables in-

terfèrent si et seulement si une définition de l'une d'entre elles est contenue dans le live-range de l'autre.”.

Question 25-5 Ces deux définitions sont différentes hors forme SSA : construire un exemple qui illustre ce point.

Question 25-6 Par contre ces deux définitions sont équivalentes sous SSA : justifiez.

Question 25-7 On généralise sans effort la notion de dominance et de graphe de dominance à l'ensemble des instructions. Montrez que l'ensemble des points où une variable est en vie est un sous-arbre de l'arbre de dominance.

26 Élimination d'expressions redondantes

Question 26-1 Donnez un algorithme linéaire qui effectue la copy-propagation sous SSA.

Question 26-2 La sémantique d'une instruction ϕ est proche de celle d'une copie. Généraliser l'algorithme donné ci-dessus pour traiter les ϕ .

Question 26-3 Donnez un exemple de copie redondante que cet algorithme ne peut éliminer.

Question 26-4 On suppose que l'on dispose d'une procédure magique nommée **simplify** qui simplifie une expression et la met sous une forme canonique. Proposer une généralisation de l'algorithme donné ci-dessus pour traiter les expressions redondantes.

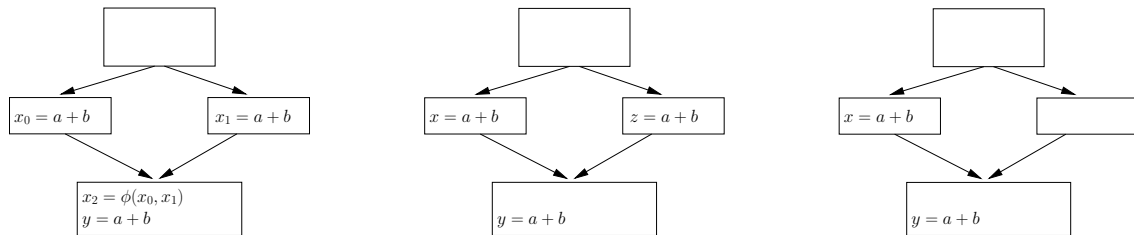


FIG. 10 – Exemple d'expressions redondantes non éliminées par du value-numbering en une passe.

Question 26-5 Peut-on améliorer l'algorithme pour traiter les exemples de la Figure 10 ?

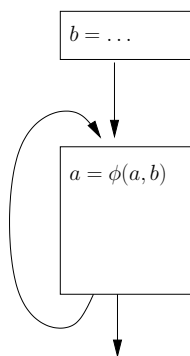


FIG. 11 – Copy redondante cyclique.

Question 26-6 Qu'en est-il de l'exemple de la Figure 11 ?

Flex (TP)

La correction de ce TP à été faite par Benoit Boissinot puis partiellement modifiée et complétée par Fabrice Rastello.

27 Un peu plus qu'un analyseur lexicographique

Le programme `flex` est un analyseur syntaxique inspiré de `lex`. On peut trouver de l'aide sur son utilisation dans les pages de manuel ou bien sur internet, notamment sur le site <http://www.gnu.org/software/flex/manual/>.

Le processus pour créer un exécutable à partir de `flex` est le suivant :

- écrire le fichier `.lex`,
- créer un fichier C, en utilisant `flex`,
- compiler le fichier C avec la bibliothèque de `flex` (`gcc machin.lex.yy.c -lfl -o machin`.
Attention à la position du `-lfl`).

Le Makefile correspondant à l'exercice est le suivant :

faire `make FOPT=-Cf` pour enlever la compression

```
all : parentheses hpcalc

parentheses hpcalc :% : %.lex.yy.c
    gcc  $< -lfl -o $@

%.lex.yy.c : %.lex
    flex ${FOPT} $<;mv lex.yy.c $@

clean :
    \rm -f *.lex.yy.c *~

cleanall : clean
    \rm -f hpcalc parentheses
```

Si vous ne connaissez pas encore `gmake` (encore un outil gnu!), il est encore temps : <http://www.gnu.org/software/make/manual/>.

La structure d'un fichier `flex` est la suivante :

```
definitions
%%
règles
%%
```

code du programme (typiquement le main)

Dans les parties *définitions* ou *règles*, le code situés entre `%{` et `%}` est incluse tel quel dans le fichier `.c` généré par `flex`, on l'utilisera par exemple pour déclarer des variables, ou inclure des bibliothèques.

Question 27-1

Écrivez un programme en `flex` qui vérifie le bon parenthésage d'un flot de données sur `stdin` : (1) 3 types de parenthèse : `()`, `[]` et `{}` ; (2) pas de retour à la ligne dans `()` ni dans `[]` ; (2) une hiérarchie dans le parenthésage, ie pas de `[]` ni `{}` dans `()`, et pas de `{}` dans `[]`.

La hiérarchie dans le parenthésage simplifie grandement l'implémentation. Le principe du programme est de compter le nombre de parenthèses (variable `p1`), crochets (variable `p2`) ou accolades (variables `p3`) déjà ouvertes, de vérifier que l'on est bien dans le bon contexte à chaque fois que l'on rencontre un nouveau symbole ("`(`", "`)`", "`[`", "`]`", "`{`", "`}`", "`\n`"), puis d'incrémenter ou décrémenter si nécessaire la variable correspondante.

```
/* contenu du fichier parentheses.lex */
%{
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>

/* () : p1
   [] : p2
   {} : p3 */

int np1=0,np2=0,np3=0;
}%

%%
"(" { np1++;}
"[" { assert(!np1); np2++; }
"{" { assert(!np1 && !np2); np3++; }
\n { assert(!np1 && !np2);}
")" { assert(np1); np1--; }
"]" { assert(np2 && !np1); np2--; }
"}" { assert(np3 && !np1 && !np2); np3--; }
.
%%

main()
{
    yylex();
    assert (!np1 && !np2 && !np3);
}
```

Question 27-2

Écrivez une calculatrice polonaise 4 opérations en `flex`. Ajoutez-y les fonction `STO` et `RCL` des calculatrices HP, avec 10 mémoires numérotées de 0 à 9.

```
%{
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>

#define S 100 /* MAXSIZE for the stack */
```

```

double pile[S];
double mem[10];
int n=-1; /* index of last element of stack */

#define s (n+1) /* size of the stack */

static void print_pile();
%}

FLOAT ("+"|"-")?[0-9]+("."[0-9]+|(e|E)("+"|"-")?[0-9]+)?

%%
{FLOAT}\n {
    double x=atof(yytext);
    assert(s<S); /* check for stack overflow
    */
    pile[++n]=x; /* increment n and save
    * the number we read
    */
}

"+" {
    assert(s>=2);
    /* decrement n, store the result */
    n--;pile[n]=pile[n]+pile[n+1];
    printf("%g",pile[n]);
}

"-" {
    assert(s>=2);
    /* decrement n, store the result */
    n--;pile[n]=pile[n]-pile[n+1];
    printf("%g",pile[n]);
}

"*" {
    assert(s>=2);
    n--;pile[n]=pile[n]*pile[n+1];
    printf("%g",pile[n]);
}

"/" {
    assert(s>=2);
    /* decrement n, store the result */
    n--;pile[n]=pile[n]/pile[n+1];
    printf("%g",pile[n]);
}

"sto" {
    int mem_id;
    assert(s>=2);
    /* decrement n, find the memory number */
    mem_id=(int) pile[n--];
    assert(mem_id>=0 && mem_id<10);
    /* decrement n, store the number at
    * the given place
    */
    mem[mem_id]=pile[n--];
}

"rcl" {

```

```

    int mem_id;
    assert(s>=1);
    /* find the memory number */
    mem_id=(int) pile[n];
    assert(mem_id>=0 && mem_id<10);
    /* write the memory in the stack */
    pile[n]=mem[mem_id];
}
%%

main()
{
    yylex();
}

```

28 Compression d'automates

Question 28-1

Compilez le fichier `.lex` de votre polonaise inversée avec l'option `-Cf` de `flex`, qui ne compresse pas les tables.

Répondez, au vu du fichier `lex.yy.c` obtenu, aux questions suivantes (qui servent à montrer pourquoi `flex` compresse les tables) :

- Combien d'états a l'automate ?
- Combien d'états sont acceptants ?
- Que représentent les nombres négatifs dans la table de transition ?
- Quelle est la taille de l'ensemble des tables en octets ?
- Quel est à vue de nez le pourcentage de transitions vers échec ?

Voici une partie de la table de transition lorsque l'option `-Cf` a été utilisée :

```

static yyconst short yy_nxt[][128] =
{
{
    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,

    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,
    0,    0,    0,    0,    0,    0,    0,    0,    0,    0,

},

{
    3,    4,    4,    4,    4,    4,    4,    4,    4,    4,
    4,    4,    4,    4,    4,    4,    4,    4,    4,    4,
    4,    4,    4,    4,    4,    4,    4,    4,    4,    4,
    4,    4,    4,    4,    4,    4,    4,    4,    4,    4,
    4,    4,    5,    6,    4,    7,    4,    8,    9,    9,
    9,    9,    9,    9,    9,    9,    9,    9,    4,    4,
    4,    4,    4,    4,    4,    4,    4,    4,    4,    4,

```



```

        4,    4,    4,    4,    4,    4,    4,    4,    4,    4,
        4,    4,    4,    4,    4,    4,    4,    4,    4,    4,
        4,    4,    4,    4,    4,    4,    4,    4,    4,    4,
        4,    4,    4,    4,    4,    4,    4,    4,    4,    4,
        4,    4,    4,    4,    10,   11,    4,    4,    4,    4,
        4,    4,    4,    4,    4,    4,    4,    4
    },
[snip]
    {
        3,   -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,
       -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,
       -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,
       -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,

       -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,
       -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,
       -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,
       -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,
       -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,
       -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,
       -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,
       -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,
       -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,  -22,
    },
} ;

```

128 ne correspond pas au nombre d'états, mais bien au nombre de caractères ASCII reconnus. Il y a 23 tables de 128 caractères correspondant aux 23 états numérotés de 0 à 22.

Voici les états d'acceptation (un état est un état d'acceptation s'il est différent de 0).

```

static yyconst short int yy_accept[23] =
{
    0,
    0,    0,    9,    8,    4,    2,    3,    5,    8,    8,
    8,    0,    1,    0,    0,    0,    0,    0,    0,    0,
    7,    6
} ;

```

Et la boucle principale où on fait tourner l'automate :

```

yy_current_state = yy_start;
yy_match:
while ( (yy_current_state = yy_nxt[yy_current_state][YY_SC_TO_UI(*yy_cp)]) > 0 )
{
    if ( yy_accept[yy_current_state] )
    {
        yy_last_accepting_state = yy_current_state;
        yy_last_accepting_cpos = yy_cp;
    }

    ++yy_cp;
}

yy_current_state = -yy_current_state;

```

Dans la table de transition, les nombres négatifs représentent le puit. De plus on peut retrouver l'état précédent, car $previous_state = -next_state$.


```

yy_current_state = (int) yy_def[yy_current_state];
if ( yy_current_state >= 24 )
yy_c = yy_meta[(unsigned int) yy_c];
}
yy_current_state = yy_nxt[yy_base[yy_current_state] + (unsigned int) yy_c];
++yy_cp;
}
while ( yy_base[yy_current_state] != 38 );

```

Question 28-3

Proposez une autre méthode de compression d'automates.

A faire.

29 Autres outils

Question 29-1

Reprogrammer votre calculatrice en perl

Le principe est exactement le même que la calculatrice en lex.
Voici le contenu du fichier :

```

#!/usr/bin/perl
use Switch;

@pile=();
@mem=();
while (<STDIN>) {
    chomp $_;
    SWITCH : {
        /^(\\+|-)?[0-9]+(\\. [0-9]+|(e|E)(\\+|\\-)?[0-9]+)?$/ && do {
            push @pile, $_;
            last SWITCH;
        };
        /^\\+|\\-|\\*|\\/$/ && do {
            $a=pop @pile; $b=pop @pile;
            $res=eval $b . $_ . $a;
            push @pile, $res;
            print $res, "\\n";
            last SWITCH;
        };
        /^sto$/ && do {
            $mem_id=pop @pile;
            ($mem_id<10 && $mem_id>=0) || die "$mem_id out of range";
            $mem[$mem_id]=pop @pile;
            last SWITCH;
        };
        /^rcl$/ && do {
            $mem_id=pop @pile;
            ($mem_id<10 && $mem_id>=0) || die "$mem_id out of range";
            defined($mem[$mem_id]) || die "mem[$mem_id] undefined";
            push @pile, $mem[$mem_id];
            last SWITCH;
        };
    };
};

```

```

/^p/ && do {
    print join(" ",@pile),"\n";
    last SWITCH;
};

die "hum, command unknown";
}
}

```

On remarquera la facilité d'implémentation en perl due :

- La manipulation simple de structures dynamiques complexes (pile, table de hashage, etc.).
- La puissance expressive des expressions régulières (RegExp) en perl.
- La surcharge des opérateurs et la complexité du typage.
- L'existence d'un nombre effroyable de bibliothèques implémentées en perl
- etc.

En résumé, même si ce petit exercice n'est pas très probant pour prouver l'utilité du perl, si vous ne connaissez pas le perl, ne tardez plus à l'apprendre !

Par contre, un programme perl est rapidement illisible.

Question 29-2

Il y a dans la distribution source d'Objective Caml un fichier `lex/compact.ml`. Quel est son principe ?

On peut trouver le fichier en question en ligne, notamment à l'adresse <http://www.mit.edu/afs/sipb.mit.edu/project/ocaml/src/current/lex/compact.ml>.

Bison (TP)

La correction de ce TD à été faite par Julien Robert à partir des implémentations de Fabrice Rastello.

Googélisez avec les mots clés **bison** & **manual**. Récupérez la documentation. Essayez **man** & **info**.

30 Une somme avec associativité à droite

Question 30-1

Soit la grammaire suivante avec associativité à droite :

$$\begin{array}{lcl} \text{sum} & \rightarrow & \text{integer } \pm \text{ sum} \\ & | & \text{integer} \end{array}$$

Implémentez en bison cette grammaire. Utilisez flex pour générer les tokens integer & \pm ou écrivez la fonction `yylex()` directement en c.

Tout d'abord, on va écrire l'analyseur grammatical en bison (`sum_f.l.y`) :

```
%{
int yyerror(char *s);
}%

%token INTEGER

%%
input:      /* empty */
| input line
;

line:      '\n'
| sum '\n' { printf ("\t%d\n", $1); }
;

sum:      INTEGER          { $$ = $1;          }
| sum '+' INTEGER        { $$ = $1 + $3;      }
;
%%

#include <stdio.h>

yyerror (char *s) /* Called by yyparse on error */
{
    printf ("%s\n", s);
}
```

```

}

int main ()
{
    return yyparse ();
}

```

Attention, le programme bison ne fait pas le travail de lex, donc ce programme ne suffit pas, il faut parser la chaîne lue et la renvoyer sous forme de “Tokens”, c’est à dire, ici INTEGER et ‘+’. La fonction effectuant ce travail doit se nommer `yylex()`. Dans notre cas, on lira la chaîne de caractères sur l’entrée standard. On doit renvoyer des Tokens et mettre leur valeurs à dans `yyval`.

Première méthode : codage de la fonction `yylex` directement (à inclure dans le fichier bison)

```

yylex ()
{
    int c;

    /* skip white space */
    while ((c = getchar ()) == ' ' || c == '\t')
        ;
    /* process numbers */
    if (isdigit (c))
    {
        ungetc (c, stdin); //Push c dans le flot stdin: revient en arriere.
        scanf ("%d", &yyval);
        return INTEGER;
    }
    /* return end-of-file */
    if (c == EOF)
        return 0;
    /* return single chars */
    return c;
}

```

Cette méthode a le mérite d’être rapide à coder si on ne connaît pas flex, mais grâce au TP précédent on maîtrise parfaitement flex et on peut donc écrire notre fonction de de la manière suivante(bien que ce soit, pour ce langage “la pierre pour tuer la mouche sur la tête de l’ours”) :

```

%{
#include"sum_f_1.h" //Permet de récupérer les définitions de yyval,
                    //des tokens, etc.
}%

%%
[0-9]+ { yyval = atoi(yytext); return INTEGER;}
"\n"   { return '\n';}
"+"    { return '+';}
[ ]
%%

```

Pour compiler la partie bison, la commande “`bison monfichier.y`” génère le fichier “`monfichier.tab.c`” qu’il suffit de compiler avec gcc. Lorsqu’on utilise flex, c’est un peu plus compliqué, le mieux est de créer un Makefile ayant la forme suivante :

```

flex_exec=sum_f_1
flex_yo=$(flex_exec:=.y.o)

```

```

flex_yh=$(flex_exec=.h)
flex_fo=$(flex_exec=.lex.o)

all: $(flex_exec)

$(hand_exec): %: %.tab.c
    gcc $< -o $@

$(hand_yc): %.tab.c: %.y
    bison $<

$(flex_exec): %: %.y.o %.lex.o
    gcc $? -o $@ -lfl

$(flex_yo): %.y.o: %.y
    bison $*.y
    gcc -c $*.tab.c -o $*.y.o
    rm -f $*.tab.c

$(flex_yh): %.h: %.y
    bison -d $< -o $*.c

$(flex_fo): %.lex.o: %.lex %.h
    flex $*.lex
    gcc -c lex.yy.c -o $*.lex.o
    rm -f lex.yy.c

```

Question 30-2

Implémentez la version associative à gauche.

Donnez à manger une longue somme à vos deux calculatrices (e.g. `perl -e 'print "0";for my i (0 .. 100000) {print "+",int(rand(10))}'`).

Il y a deux manières d'écrire la version associative à gauche, la première consiste à le faire manuellement en modifiant un peu la grammaire précédente :

```

%token INTEGER

%%
input:      /* empty */
           | input line
;

line:       '\n'
           | sum'\n'  { printf ("\t%d\n", $1); }
;

sum:        INTEGER          { $$ = $1;          }
           | INTEGER '+' sum { $$ = $1 + $3;      }
;
%%

```

La seconde méthode utilise le fait que l'on peut préciser à bison si un token doit être associatif à gauche ou à droite :

```

%token INTEGER
%left '+'      //C'est ici que l'on précise que la règle doit etre
               //associative à gauche.

%%

```

```

input:      /* empty */
          | input line
;

line:       '\n'
          | sum '\n' { printf ("\t%d\n", $1); }
;

sum:        INTEGER      { $$ = $1;      }
          | sum '+' sum  { $$ = $1 + $3;  }
;
%%

```

La grammaire associative à droite implique un empilement des tokens, ce qui, dans le cas d'une grande entrée fait déborder la pile (ceci se traduit par le message d'erreur : "parser stack overflow").

31 Evaluation d'expression

Question 31-1

Reprenez dans votre cours la grammaire désambiguïsée des expressions arithmétiques usuelles. Implémentez la.

De la même manière que précédemment, on écrit deux fichiers :

expr.y :

```

%{
#define YYSTYPE double
int yyerror(char *s);
%}

%token FLOAT

%%
input:      /* empty */
          | input line
;

line:       '\n'
          | exp '\n' { printf ("\t%f\n", $1); }
;

exp:  exp '+' term      { $$ = $1 + $3;  }
    | exp '-' term      { $$ = $1 - $3;  }
    | term              { $$ = $1;      }
;

term:  term '*' factor   { $$ = $1 * $3;   }
    | term '/' factor   {
                        if ($3==0) {
                            int i;
                            for (i=0;i<=@2.first_column;i++) putchar(' ');
                            printf("^ division par zero\n");
                            exit(1);}
                        else $$ = $1 / $3;
                    }
    | factor            { $$ = $1;      }
;

```



```

factor: '(' exp ')' { $$ = $2; }
      | FLOAT      { $$ = $1; }
;
%%

#include <stdio.h>

yyerror (char *s) /* Called by yyparse on error */
{
    printf ("%s\n", s);
}

int main ()
{
    return yyparse ();
}

expr.lex :

%{
#define YYSTYPE double //Attention,, l'ordre de ces lignes est important!
#include "exp.y.h"
%}

NUMBER [0-9]+("."[0-9]+|(e|E)( "+" | "-" )?[0-9]+)?

%%
{NUMBER}          {
                    yyval = atof(yytext);
                    DEBUG(printf("lexparse %f\n",yyval);)
                    return FLOAT;}
"\n"              {
                    DEBUG(printf("lexparse \n\n");)
                    return '\n';}
" + " | " - " | " * " | " / " | " ( " | " ( " {
                    DEBUG(printf("lexparse %c\n",yytext[0]);)
                    return yytext[0];}
[ ]
%%

```

Question 31-2

Lorsqu'une opération génère une erreur (e.g. division par 0) affichez une erreur qui localise l'opérateur fautif (e.g. \wedge sous l'expression). Utilisez la notion de location maintenue par **bison**.

La gestion de la division par zero se fait dans le programme bison, on maintient dans le programme la connaissance de l'endroit où on se trouve grâce à `yyloc` : dans `yyloc.first_column`, on indique combien de caractères ont été lus.

`exp.y` :

```

%{
#define YYSTYPE double
int yyerror(char *s);
%}

%token FLOAT

%%
input:      /* empty */

```

```

        | input line
;

line:      '\n'
        | exp '\n' { printf ("\t%f\n", $1); }
;

exp:      exp '+' term      { $$ = $1 + $3;    }
        | exp '-' term      { $$ = $1 - $3;    }
        | term              { $$ = $1;        }
;
term:      term '*' factor   { $$ = $1 * $3;    }
        | term '/' factor   {
                                if ($3==0) {
                                    int i;
                                    for (i=1;i<@3.first_column;i++) putchar(' ');
                                    printf("^ division par zero\n");
                                    exit(1);}
                                else $$ = $1 / $3;
                                }
        | factor            { $$ = $1;        }
;
factor:    '(' exp ')' { $$ = $2; }
        | FLOAT        { $$ = $1; }
;
%%

#include <stdio.h>

yyerror (char *s) /* Called by yyparse on error */
{
    printf ("%s\n", s);
}

int main ()
{
    return yyparse ();
}

exp.lex :

%{
#include "exp.h"
#include <ctype.h>
#define YYSTYPE double
%}

NUMBER [0-9]+(("[0-9]+|(e|E)( "+"| "-" )?[0-9]+)?

%%
{NUMBER}
{
    int i;
    yylval = atof(yytext);
    for(i = 0;yytext[i];i++)
        ++yylloc.first_column;
    return FLOAT;}

"\n"
{

```

```

       yyval.first_column=0;
        return '\n';}

"+"|"-"|"*"|"/"|"")"|"(" {
    ++yyval.first_column;
    return yytext[0];}

" " { ++yyval.first_column;} //Il faut compter
                                     //les espaces

%%

```

Spim (TP)

La correction de ce TP à été faite par Nada Ahmidani et Ulrich Herberg à partir des implémentations de Tanguy Risset.

32 Mise en place

Nous allons utiliser le simulateur Spim (<http://www.cs.wisc.edu/~larus/spim.html>) développé par James Larus professeur à l'université du Wisconsin. Spim est un simulateur du jeu d'instruction du Mips (*instruction set simulator* : ISS). La description du jeu d'instruction du Mips est disponible ici : `/home/trisсет/src/HP_AppA.pdf`, une documentation du logiciel `xspim` (version X de Spim) est disponible ici : `/home/trisсет/src/spim_documentation.pdf`

Pour produire du code assembleur Mips, nous allons utiliser GCC, reciblé pour la plate-forme Mips. Le reciblage de GCC se fait lors de la compilation de `gcc`, j'ai donc recompilé GCC pour Mips.

Question 32-1

Le binaire reciblé pour Mips est `mipsel-linux-elf-gcc`. Pour l'utiliser il faut rajouter ce chemin à votre `$PATH`

On exécute la commande :

```
export PATH=/home/trisсет/cxtools/bin/:$PATH
```

Question 32-2

Pour utiliser Spim (commande `xspim`) rajouter à votre `$PATH` le chemin `/home/trisсет/bin`

On exécute la commande :

```
export PATH=/home/trisсет/bin/:$PATH
```

Question 32-3

Récupérez le fichier `/home/trisсет/src/fib.c` contenant une fonction calculant la suite de Fibonacci

On exécute la commande :

```
cp /home/trisсет/src/fib.c
```

33 Utilisation de Spim

Question 33-1

Générez un fichier assembleur `fib.s` Mips à partir du fichier `fib.c` (options `-S` pour GCC) :
`mipsel-linux-elf-gcc -S fib.c`.
 Une version commentée de ce fichier se trouve ici :
`/home/trisсет/src/fib-com.s`.

```
mipsel-linux-elf-gcc -S fib.c
```

Question 33-2

Lancer xspim et charger le fichier `fib.s` dans Spim. Vous allez avoir besoin d'éditer le fichier assembleur car GCC génère deux directives qui ne sont pas implémentées dans Spim : `.section` et `.previous`. Supprimez simplement ces deux lignes.

```
#Version Commentée du fichier fib.s
# les numero de registre on été remplacé par leur nom:
#par exemple $2 remplacé par $v0
.file      1 "fib.c"
.text
.align     2
.globl     fib
.ent       fib

fib:
.frame     $fp,40,$ra
.mask      0xc0010000,-8
.fmask     0x00000000,0
subu       $sp,$sp,40      # SP <- SP-40 :AR de 40 octet (10 mots)
sw         $ra,32($sp)     # stocke adresse retour SP+32
sw         $fp,28($sp)     # stocke ARP appelant SP+28
sw         $s0,24($sp)     # sauvegarde registre $s0
move       $fp,$sp        # ARP <- SP
sw         $a0,40($fp)     # stocke Arg1 dans la pile (ARP+40)
lw         $v0,40($fp)     # charge Arg1 dans $v0
slt        $v0,$v0,2       # $v0 <- 1 si $v0<2 0 sinon
beq        $v0,$0,$L2     # branch L2 si $v0==0
li         $v0,1          # $v0 <- 0x1
                        # ($v0 sera le registre contenant le res)
sw         $v0,16($fp)     # stocke le resultat dans la pile
j          $L1            # saute à L1

$L2:
lw         $v0,40($fp)     # charge Arg1 dans $v0
addu       $v0,$v0,-1     # retranche 1
move       $a0,$v0        # $a0 <- $v0
                        # ($a0 contient Arg1 pour l'appel recursif)
jal        fib            # jump and link fib ($ra<-next instr)
move       $s0,$v0        # $s0 <- $v0 ($v0: res appel fib)
lw         $v0,40($fp)     # charge Arg1 dans $v0
addu       $v0,$v0,-2     # retranche 2
move       $a0,$v0        # $a0 <- $v0
                        # ($a0: contient Arg1 pour l'appel recursif)
jal        fib            # jump and link fib ($ra<-next instr)
addu       $s0,$s0,$v0     # $s0 <- $s0+$v0 ($v0: res appel fib)
sw         $s0,16($fp)     # stocke le resultat dans la pile

$L1:
lw         $v0,16($fp)     # $v0 <- resultat
move       $sp,$fp        # SP <- ARP
lw         $ra,32($sp)     # $ra <- adresse retour
lw         $fp,28($sp)     # ARP <- ARP appelant
lw         $s0,24($sp)     # restaure $s0
addu       $sp,$sp,40     # SP->SP+40
j          $ra            # jump adresse retour
.end       fib
.align     2
```

```

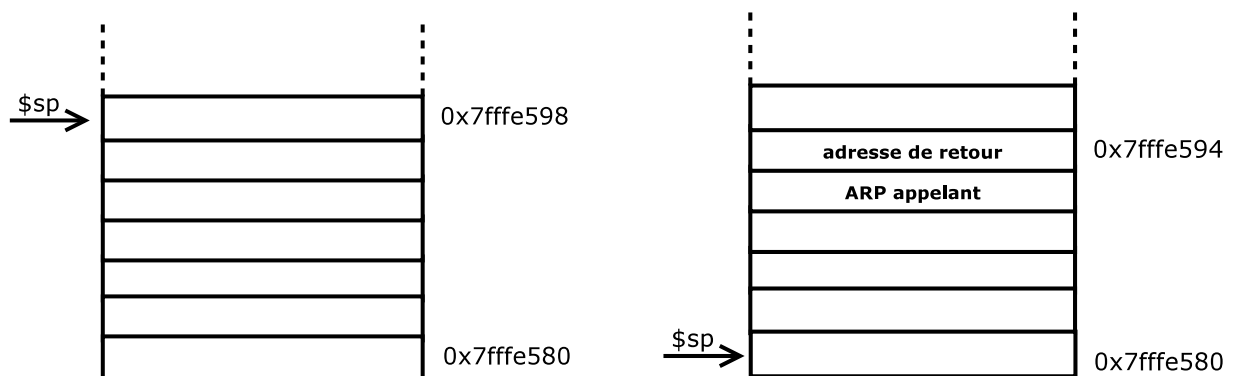
        .globl      main
        .ent        main
main:
        .frame      $fp,24,$ra    # vars= 0, regs= 2/0, args= 16, extra= 0
        .mask       0xc0000000,-4
        .fmask      0x00000000,0
        subu        $sp,$sp,24    # SP <- SP-24 :AR de 24 octet (6 mots)
        sw          $ra,20($sp)    # stocke adresse retour SP+20
        sw          $fp,16($sp)    # stocke ARP appelant SP+16
        move        $fp,$sp       # ARP <- SP
        sw          $a0,24($fp)    # stocke Arg1 dans la pile (ARP+24)
        sw          $5,28($fp)    # stocke Arg2 dans la pile (ARP+48)
        li          $a0,2         #
                                   # $a0 <- 2 ($a0: Arg1)
        jal         fib           # jump and link fib ($ra<-next instr)
        move        $sp,$fp       # SP <- ARP
        lw          $ra,20($sp)    # $ra <- adresse retour
        lw          $fp,16($sp)    # ARP <- ARP appelant
        addu        $sp,$sp,24    # SP->SP+24
        j           $ra           # jump adresse retour
        .end        main

```

Question 33-3

Exécutez pas à pas votre programme et dessinez la pile, jusqu'au premier appel récursif de `fib`.

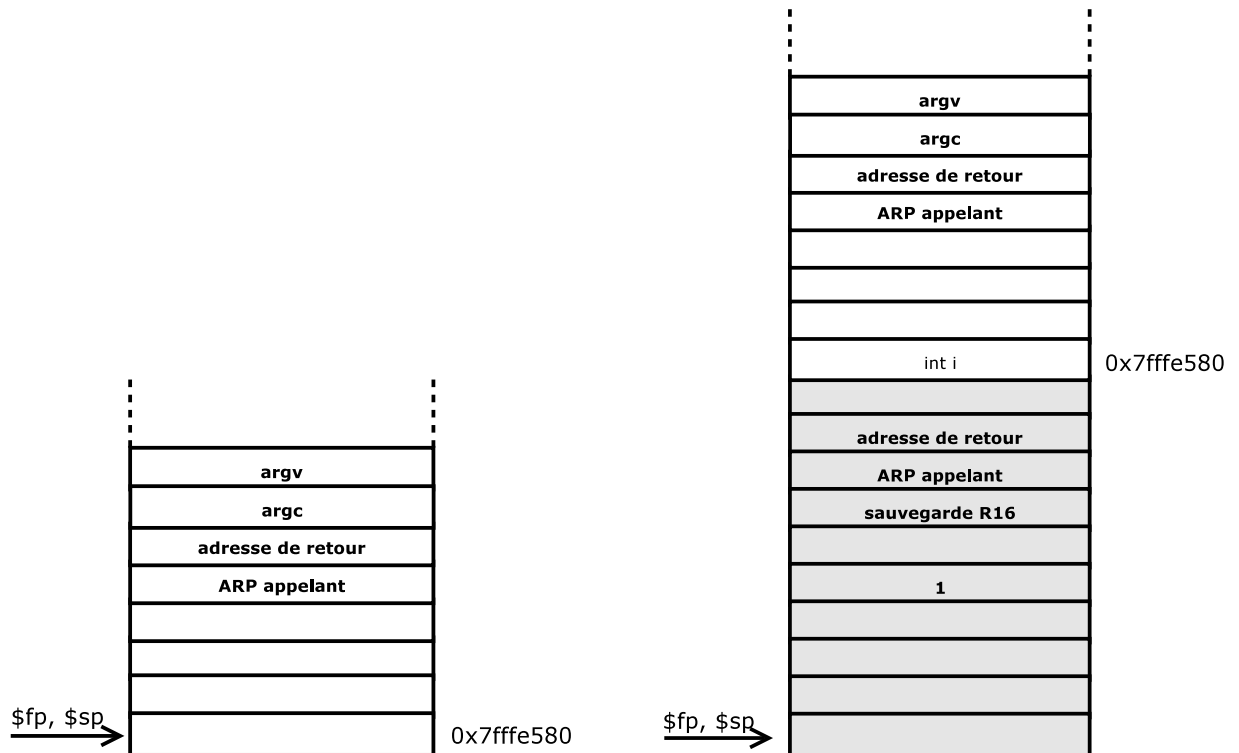
Voici les dessins de la pile après l'exécution des commandes ci-dessous



```

subu $sp, $sp, 24
sw $31,20($sp)
sw $fp,16($sp)

```



```

move $sp, $fp
sw $4, 24($fp)
sw $5, 28($fp)

```

```

subu $sp, $sp, 40
sw $31, 32($sp)
sw $fp, 28($sp)
sw $16, 24($sp)
sw $4, 40($sp)
sw $2, 16($sp)

```

34 Recompilation de Spim

Nous allons modifier Spim pour qu'il accepte les directives inconnues. La compilation de Spim utilise **bison** pour générer le parser. l'analyseur lexical est dans le fichier **scanner.l**, l'analyseur syntaxique est dans le fichier **parser.y** (on n'essayera pas de comprendre tout). Les mots clés du langage assembleur sont déclarés dans le fichier **op.h** sous la forme : **OP(NAME, OP_CODE, TYPE, R_OPCODE)**. Par exemple la directive **.data** est déclaré par la ligne : **OP(".data", Y_DATA_DIR, ASM_DIR, -1)**. Cette déclaration est utilisée pour décrire les tokens renvoyés par l'analyseur lexical : lorsque l'analyseur lexical rencontre la directive **.data**, il renvoi le token **Y_DATA_DIR**.

Question 34-1

Récupérer le source de Spim en **/home/trisсет/src/spim.tar**. Recompiler Spim sans le modifier : **make clean; make**. Que remarquez vous concernant le parseur de Spim ?

```

cp /home/trisсет/src/spim.tar .
tar xvf spim.tar
cd spim-7.0
make clean; make

```

Il y a des conflits **shift / reduce** dans le parseur.

Rajouter les deux directives inconnues de Spim **.section** et **.previous** dans le parseur (le parseur ne fera aucune action sur ces directives). Pour cela on effectue les actions suivantes :

Question 34-2

Ajouter la déclaration des tokens correspondant aux nouvelles directives dans le fichier **op.h**. Attention, les déclarations du fichier **op.h** doivent être faites dans l'ordre alphabétique.

```

OP(".align",      Y_ALIGN_DIR,      ASM_DIR,      -1)
OP(".ascii",     Y_ASCII_DIR,      ASM_DIR,      -1)
OP(".asciiz",    Y_ASCIIZ_DIR,    ASM_DIR,      -1)
OP(".asm0",      Y_ASMO_DIR,      ASM_DIR,      -1)
OP(".bgnb",      Y_BGNB_DIR,      ASM_DIR,      -1)
OP(".byte",      Y_BYTE_DIR,      ASM_DIR,      -1)
OP(".comm",      Y_COMM_DIR,      ASM_DIR,      -1)
OP(".data",      Y_DATA_DIR,      ASM_DIR,      -1)
OP(".double",    Y_DOUBLE_DIR,    ASM_DIR,      -1)
OP(".end",       Y_END_DIR,      ASM_DIR,      -1)
OP(".endb",      Y_ENDB_DIR,      ASM_DIR,      -1)
OP(".endr",      Y_ENDR_DIR,      ASM_DIR,      -1)
OP(".ent",       Y_ENT_DIR,      ASM_DIR,      -1)
OP(".err",       Y_ERR_DIR,      ASM_DIR,      -1)
OP(".extern",    Y_EXTERN_DIR,    ASM_DIR,      -1)
OP(".file",      Y_FILE_DIR,      ASM_DIR,      -1)
OP(".float",     Y_FLOAT_DIR,     ASM_DIR,      -1)
OP(".fmask",     Y_FMASK_DIR,     ASM_DIR,      -1)
OP(".frame",     Y_FRAME_DIR,     ASM_DIR,      -1)
OP(".globl",     Y_GLOBAL_DIR,    ASM_DIR,      -1)
OP(".half",      Y_HALF_DIR,      ASM_DIR,      -1)
OP(".kdata",     Y_K_DATA_DIR,    ASM_DIR,      -1)
OP(".ktext",     Y_K_TEXT_DIR,    ASM_DIR,      -1)
OP(".lab",       Y_LABEL_DIR,     ASM_DIR,      -1)
OP(".lcomm",     Y_LCOMM_DIR,     ASM_DIR,      -1)
OP(".liverereg", Y_LIVEREG_DIR,    ASM_DIR,      -1)
OP(".loc",       Y_LOC_DIR,      ASM_DIR,      -1)
OP(".mask",      Y_MASK_DIR,      ASM_DIR,      -1)
OP(".noalias",   Y_NOALIAS_DIR,   ASM_DIR,      -1)
OP(".option",    Y_OPTIONS_DIR,   ASM_DIR,      -1)
OP(".previous",  Y_PREVIOUS_DIR,  ASM_DIR,      -1)
OP(".rdata",     Y_RDATA_DIR,     ASM_DIR,      -1)
OP(".repeat",    Y_REPEAT_DIR,    ASM_DIR,      -1)
OP(".sdata",     Y_SDATA_DIR,     ASM_DIR,      -1)
OP(".section",   Y_SECTION_DIR,   ASM_DIR,      -1)
OP(".set",       Y_SET_DIR,       ASM_DIR,      -1)
OP(".space",     Y_SPACE_DIR,     ASM_DIR,      -1)
OP(".struct",    Y_STRUCT_DIR,    ASM_DIR,      -1)
OP(".text",      Y_TEXT_DIR,      ASM_DIR,      -1)
OP(".verstamp",  Y_VERSTAMP_DIR,  ASM_DIR,      -1)
OP(".vreg",      Y_VREG_DIR,      ASM_DIR,      -1)
OP(".word",      Y_WORD_DIR,      ASM_DIR,      -1)
OP(".abs",       Y_ABS_POP,       PSEUDO_OP,     -1)

```

Question 34-3

Modifier le fichier `parser.y` en déclarant les nouvelles directives et en ajoutant les deux règles pour les reconnaître

```
/* Assembler directives: */
```

```

%token Y_ALIAS_DIR
%token Y_ALIGN_DIR
%token Y_ASCII_DIR
%token Y_ASCIIZ_DIR
%token Y_ASMO_DIR

```



```

%token Y_BGNB_DIR
%token Y_BYTE_DIR
%token Y_COMM_DIR
%token Y_DATA_DIR
%token Y_DOUBLE_DIR
%token Y_END_DIR
%token Y_ENDB_DIR
%token Y_ENDR_DIR
%token Y_ENT_DIR
%token Y_ERR_DIR
%token Y_EXTERN_DIR
%token Y_FILE_DIR
%token Y_FLOAT_DIR
%token Y_FMASK_DIR
%token Y_FRAME_DIR
%token Y_GLOBAL_DIR
%token Y_HALF_DIR
%token Y_K_DATA_DIR
%token Y_K_TEXT_DIR
%token Y_LABEL_DIR
%token Y_LCOMM_DIR
%token Y_LIVEREG_DIR
%token Y_LOC_DIR
%token Y_MASK_DIR
%token Y_NOALIAS_DIR
%token Y_OPTIONS_DIR
  %token Y_PREVIOUS_DIR
%token Y_RDATA_DIR
%token Y_REPEAT_DIR
%token Y_SDATA_DIR
  %token Y_SECTION_DIR
%token Y_SET_DIR
%token Y_SPACE_DIR
%token Y_STRUCT_DIR
%token Y_TEXT_DIR
%token Y_VERSTAMP_DIR
%token Y_VREG_DIR
%token Y_WORD_DIR

/* les nouvelles directives */

ASM_DIRECTIVE:  Y_ALIAS_DIR      Y_REG  Y_REG

               |      Y_ALIGN_DIR  EXPR
               {
                 align_data ($2.i);
               }

/* etc..... quelques lignes omis */

               |      Y_OPTIONS_DIR  ID

               /      Y_PREVIOUS_DIR

```

```

|      Y_REPEAT_DIR      EXPR
|      {
|          yyerror ("Warning: repeat directive ignored");
|      }

|      Y_RDATA_DIR
|      {
|          user_kernel_data_segment (0);
|          data_dir = 1; text_dir = 0;
|          enable_data_alignment ();
|      }

|      Y_RDATA_DIR      Y_INT
|      {
|          user_kernel_data_segment (0);
|          data_dir = 1; text_dir = 0;
|          enable_data_alignment ();
|          set_data_pc ($2.i);
|      }

|      Y_SDATA_DIR
|      {
|          user_kernel_data_segment (0);
|          data_dir = 1; text_dir = 0;
|          enable_data_alignment ();
|      }

|      Y_SDATA_DIR      Y_INT
|      {
|          user_kernel_data_segment (0);
|          data_dir = 1; text_dir = 0;
|          enable_data_alignment ();
|          set_data_pc ($2.i);
|      }

/      Y_SECTION_DIR      ID

|      Y_SET_DIR      ID
|      {
|          if (streq ((char*)$2.p, "noat"))
|              noat_flag = 1;
|          else if (streq ((char*)$2.p, "at"))
|              noat_flag = 0;
|      }

/* etc..... */

```

Question 34-4

Recompiler xspim et tester xspim sur le fichier fib.s original.

```
make xspim
./xspim
```

35 Appel système pour afficher une valeur

Spim propose un service ressemblant aux appels systèmes d'un système d'exploitation standard : affichage élémentaire sur une console et allocation mémoire (voir la documentation de Spim, page 8). Pour demander un service, un programme charge le code de l'appel système dans le registre `$v0` (code 1 pour afficher un entier, code 4 pour afficher une chaîne de caractère), met en place les arguments dans les registres `$a0-$a4` (pour l'affichage d'un entier, on charge l'entier dans le registre `$a0`). et exécute la l'instruction `syscall`.

Question 35-1

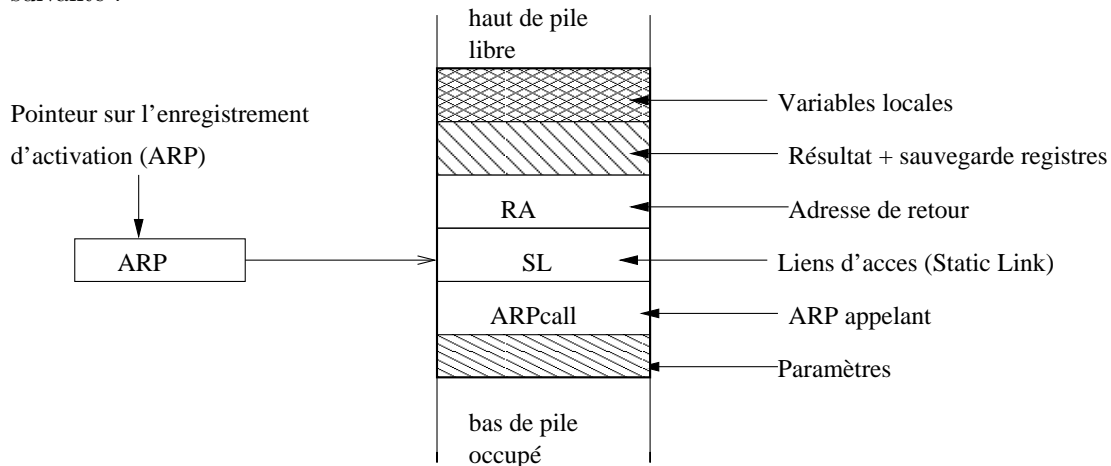
Modifier le code assembleur du fichier `fib.s` pour ajouter l'affichage de la valeur résultat après l'appel à `fib` depuis `main`.

```
move    $a0, $2      # l'argument du syscall est le résultat de fib
li      $v0, 1        # charge le code de 'print_int' dans le registre $v0
syscall                                # exécute le syscall
```

Correction de l'examen 2003-2004

36 Procédure

On représente graphiquement l'enregistrement d'activation (AR) d'une procédure de la manière suivante :



On s'intéresse donc essentiellement aux informations du lien d'accès (*Access Link, Static Link*), à l'ARP appelant (*Activation record pointer, frame pointer*) et à l'adresse de retour pointant sur le code de la procédure ayant effectué l'appel. La pile grandit vers le haut de la feuille.

À un instant donné de l'exécution du programme, la pile se retrouve dans l'état représenté sur la figure 12. On a représenté graphiquement les valeurs des pointeurs ARPCall et SL et on a indiqué explicitement par **retour vers Proc** de quelle procédure venait l'appel.

Question 36-1

Déduire de l'état de la pile, l'imbrication des procédures *main*, *X*, *Y*, *W*, *Q* (*Q* étant celle dont l'AR se trouve en sommet de pile). Utiliser un pseudo-code correctement indenté pour votre réponse. Donner aussi la branche de l'arbre d'appel correspondant à l'état de la pile représenté (i.e. qui appelle qui pour cette exécution). Justifier vos réponses.

L'ordre des AR sur la figure (de bas en haut) est : *main*, *W*, *Y*, *X*, *X*, *X*, *Q*. Donc les informations sur les appels sont *main* appelle *W* qui appelle *Y* qui appelle *X* qui s'appelle récursivement et qui appelle aussi *Q*.

D'après les liens d'accès : *X* et *Y* sont définies dans *W*. *W* et *Q* sont définies dans *main*, soit le schéma suivant :

```

Define main()
  Define Q()
  ...

```

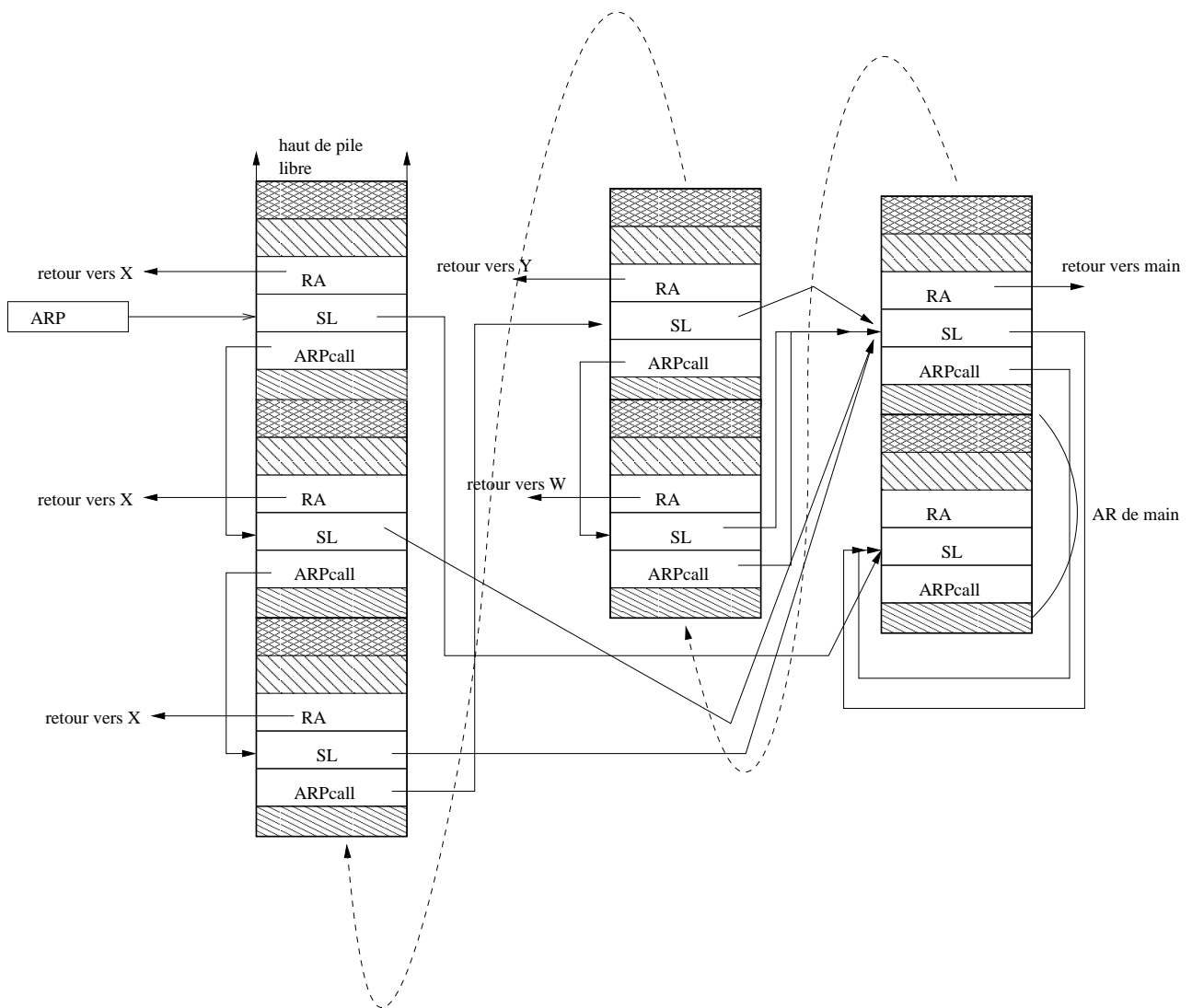


FIG. 12 – État de la pile au cours de l'exécution. On a séparé la pile en trois morceaux qui sont évidemment consécutifs

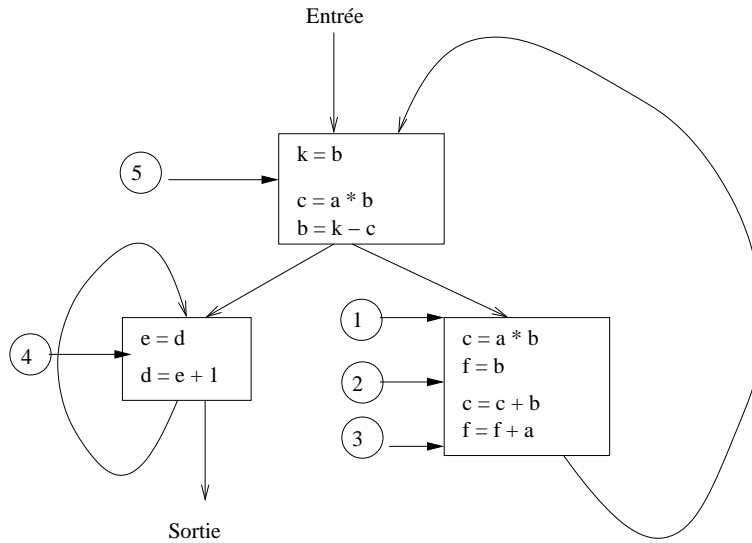


FIG. 13 – Graphe de contrôle de flot utilisé dans l'exercice 37

```

begin
end
Define W()
  Define X()
    ...
    begin          //begin X
      call X()
      call Q()
    end
  Define Y()
    ...
    begin          //begin Y
      call X()
    end
  begin          //begin W
    call Y()
  end
begin          //begin main
  call W()
end

```

37 Allocation de registres

On considère le programme dont le graphe de contrôle de flot est représenté en figure 13. On considère qu'il n'y a plus aucune variable vivante en sortie.

Question 37-1

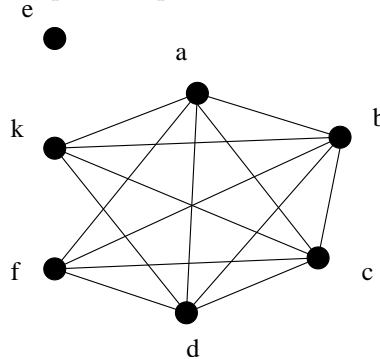
Donner les variables vivantes en chacun des point 1 à 5 du programme.

$Live(1) = \{a, b, d\}$
 $Live(2) = \{a, b, c, d, f\}$
 $Live(3) = \{a, b, d\}$
 $Live(4) = \{e\}$
 $Live(5) = \{a, b, d, k\}$

Question 37-2

Construisez le graphe d'interférences entre variables. Il s'agit du graphe dont les noeuds représentent les variables et les arcs représentent les interférences entre variables. Expliquez la différence avec le graphe d'interférence si le code était en SSA.

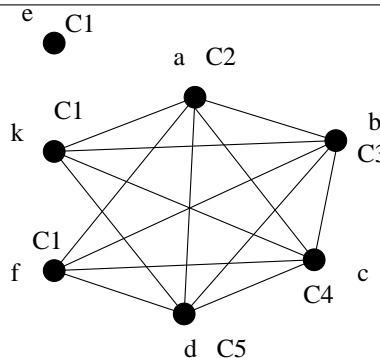
Corrigé : CS 164, Handout 23, printemps 2000



En cours on a introduit des interférences entre les durées de vie (comme si le code était en SSA). Ici, la variable c introduit deux durées de vie disjointes. En représentant ces deux durées de vie par un seul noeud (comme c'est proposé avec ce graphe), on force ces deux valeurs à être stockées dans le même registre. Ici ça ne change rien car toutes les interférences de c viennent de la deuxième durée de vie.

Question 37-3

Proposer une coloration du graphe en vue d'une allocation de registres. En déduire le nombre minimal de registres nécessaires à l'exécution du code sans recopie en mémoire.



Cinq registres sont donc nécessaires

Question 37-4

Considérons maintenant que l'on *échange* les instructions $f = b$ et $c = c + b$, c'est à dire que le contenu du bloc de base devient :

$c = a * b$

$c = c + b$

$f = b$

$f = f + a$

Quel est maintenant l'ensemble des variables vivantes avant l'instruction $f = b$? quel est maintenant le nombre de couleurs nécessaires pour colorier le graphe?

$$Live(2) = \{a, b, d\}$$

Il faut toujours 5 couleurs, il n'y a plus d'interférence entre f et c mais le groupe $\{a, b, c, d, k\}$ forme toujours une clique.

Question 37-5

L'opération d'échange ci-dessus est légale, elle ne change pas le résultat de l'exécution du code. Expliquer les conditions que doivent vérifier deux opérations successives pour être échangées :

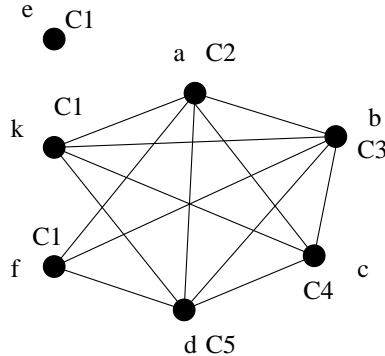
$$x_1 = y_1 \text{ op } z_1$$

$$x_2 = y_2 \text{ op } z_2$$

il faut qu'il n'y ait aucune dépendance entre les instructions : ni dépendance de données : $x_1 \neq y_2, x_1 \neq z_2$ ni dépendance de sorties : $x_1 \neq x_2$, ni anti dépendances $x_2 \neq y_1, x_2 \neq z_1$.

Question 37-6

Considérons le nouveau graphe de contrôle de flot (après échange), supposons que l'on ne dispose que de 4 registres. Quels noeuds du graphe suffit-il de colorer pour réaliser une allocation valide.



il suffit de colorer les noeuds a,b,c,d,k : les deux autres ont un degrés inférieur au nombre de registre, on pourra toujours les colorer.

Question 37-7

Une façon simple de forcer deux variables à partager le même registre est de fusionner les noeuds correspondants dans le graphe d'interférence (cela permet d'éviter les *move* superflus entre registres du type $r_x = r_y$). Expliquer dans quelles conditions il est légal de fusionner ainsi deux noeuds.

Lorsque les durées de vie des variables correspondantes sont disjointes (pas d'arc dans le graphe entre les deux noeuds).

38 Utilisation des auto-incréments

La compilation pour DSP introduit des contraintes spécifiques. Par exemple, les assembleurs sont souvent dépourvus d'adressage indirect et indirect indexé, il ne peuvent que charger le contenu d'une adresse stockée dans un registre spécifique : le registre d'adresse (noté AR dans la suite). En revanche, ils possèdent généralement des versions avec auto-incrément (ou auto-décroement) des instructions standard : *load*, *store*, *add*.

Dans cet exercice on va considérer le jeu d'instruction suivant :

<i>instruction</i>	<i>effet</i>
$AR \leftarrow val$	$AR \leftarrow val$ //chargement du registre d'adresse AR avec une valeur
$R \leftarrow *(AR)$ $R \leftarrow *(AR++)$ $R \leftarrow *(AR--)$	$R \leftarrow mem(AR)$ // AR est le registre d'adresse $R \leftarrow mem(AR); AR \leftarrow AR + 4$ //version avec auto incrément $R \leftarrow mem(AR); AR \leftarrow AR - 4$ //version avec auto decrement
$*(AR) \leftarrow R$ $*(AR++) \leftarrow R$ $*(AR--) \leftarrow R$	$mem(AR) \leftarrow R$ // AR est le registre d'adresse $mem(AR) \leftarrow R; AR \leftarrow AR + 4$ //version avec auto incrément $mem(AR) \leftarrow R; AR \leftarrow AR - 4$ //version avec auto decrement
$R_2 \leftarrow R_1 + *(AR)$ $R_2 \leftarrow R_1 + *(AR++)$ $R_2 \leftarrow R_1 + *(AR--)$ $R_3 \leftarrow R_1 + R_2$	$R_2 \leftarrow mem(AR) + R_1$ // AR est le registre d'adresse $R_2 \leftarrow mem(AR) + R_1; AR \leftarrow AR + 4$ //version avec auto incrément $R_2 \leftarrow mem(AR) + R_1; AR \leftarrow AR - 4$ //version avec auto decrement $R_3 \leftarrow R_1 + R_2;$ //version sans registre d'adresse

Avec ce jeu d'instruction, le chargement d'une variable locale d'une procédure se fait en deux instructions :

$$AR \leftarrow @a$$

$$R_1 \leftarrow *(AR)$$

Mais le chargement de trois variables rangées consécutivement dans la pile peut se faire en

quatre instructions :

$$\begin{aligned} AR &\leftarrow @a \\ R_1 &\leftarrow *(AR++) \\ R_2 &\leftarrow *(AR++) \\ R_3 &\leftarrow *(AR) \end{aligned}$$

Ce type d'optimisation dépend donc du rapport entre l'ordre dans lequel les données sont rangées en mémoire et l'ordre dans lequel elles sont accédées. Pour simplifier, on supposera que chaque opération coûte 1, il faudra donc minimiser le nombre d'opérations. On suppose aussi que l'on ne travaille qu'avec des variables locales à la procédure qui seront donc rangées dans la pile. Pour résoudre le problème d'optimisation, on procède généralement de la manière suivante :

1. On choisit d'abord un ordre d'accès aux variables à partir du code à traduire puis,
2. On essaye de placer les variables dans la pile de manière à utiliser au maximum les opérations avec auto-incrément.

Par exemple, considérons le code suivant :

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

Supposons que l'on choisisse l'ordre d'accès suivant aux différentes variables :

$$a \ b \ c \ d \ e \ f \ a \ d \ a \ d \ a \ c \ d \ f \ a \ b$$

Question 38-1

Proposer une modélisation du problème général à l'aide d'un graphe et une méthode de résolution. Résolvez le problème pour cet exemple particulier, c'est à dire rangez les 6 variables dans la pile dans un ordre qui minimise le nombre total d'instructions assembleur nécessaires pour le code.

Cet exercice est tiré du bouquin de fisher "Embedded Computing : A VLIW Approach to Architecture, Compilers and Tools" qui devrait sortir bientôt.

On construit un *access graph* dont les noeuds sont les variables et les arcs sont étiquetés par le nombre de fois que les variables sont voisines dans la séquence. Par exemple pour notre séquence, on va construire un (figure non faite, cf Fisher p 341).

On va ensuite chercher un chemin hamiltonien dans le graphe (en utilisant éventuellement des arcs inexistant qui compteront zero) en essayant de maximiser le poids. Apparemment c'est NP complet d'après Fisher. exemple, l'ordre $e \ f \ a \ d \ c \ b$ donne un total de 10 accès voisins.

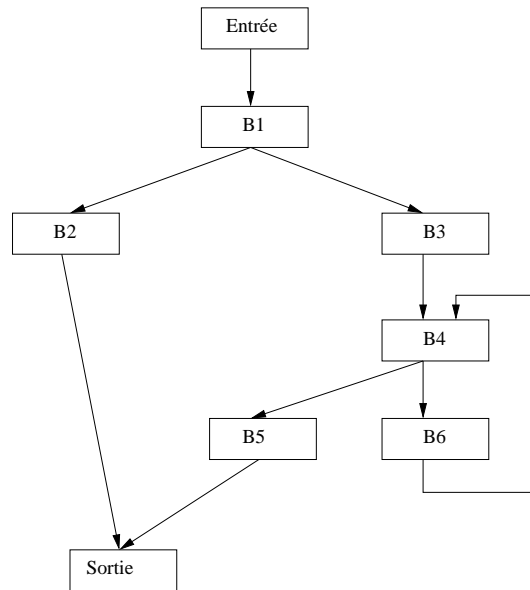
Question 38-2

Donner le code produit avec les instructions assembleur ci-dessus.

à faire, cf Fisher p341

39 Dominateurs

Considérons le graphe de contrôle de flot suivant :



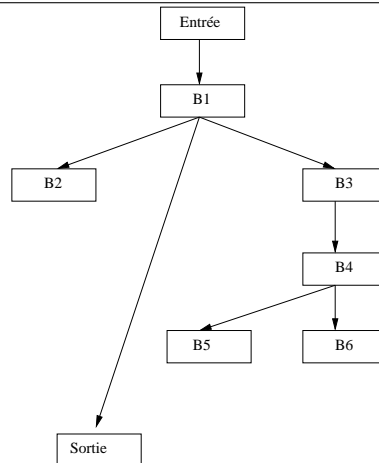
Question 39-1

Donner pour chacun des blocs, l'ensemble des dominateurs du blocs.

$Dom(Entrée) = \{Entrée\}$
 $Dom(B_1) = \{Entrée, B_1\}$
 $Dom(B_2) = \{Entrée, B_1, B_2\}$
 $Dom(B_3) = \{Entrée, B_1, B_3\}$
 $Dom(B_4) = \{Entrée, B_1, B_3, B_4\}$
 $Dom(B_5) = \{Entrée, B_1, B_3, B_4, B_5\}$
 $Dom(B_6) = \{Entrée, B_1, B_3, B_4, B_6\}$
 $Dom(Sortie) = \{Entrée, B_1, Sortie\}$

Question 39-2

Donner l'arbre de domination.



Question 39-3

Rappeler la définition de post-dominance et donner l'arbre de post-dominance.

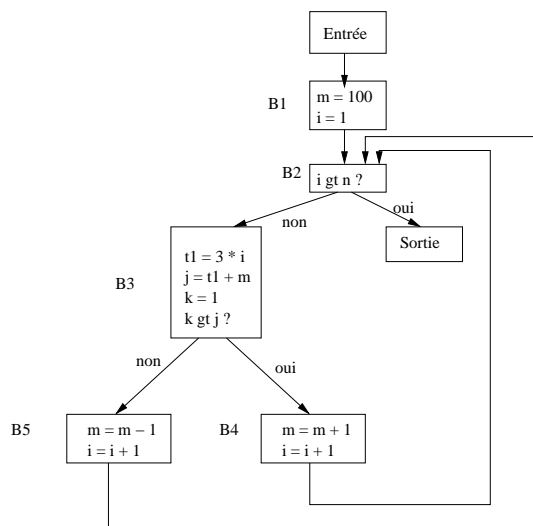
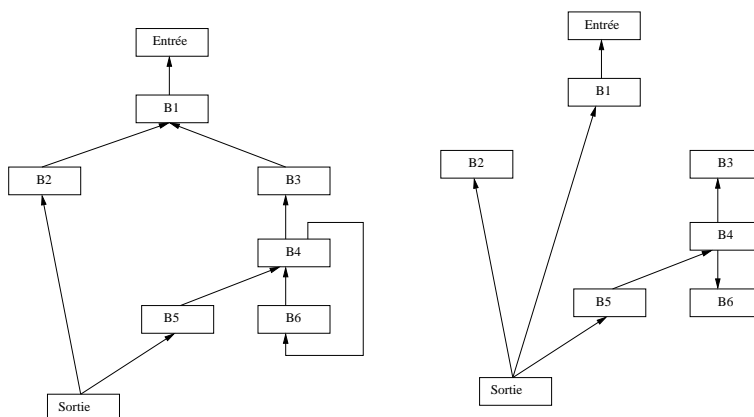


FIG. 14 – Graphe de contrôle de flot d'un programme contenant une unique boucle



Question 39-4

Rappeler la définition de dépendance de contrôle. Donner pour les noeuds desquels le noeud B_5 dépend du point de vue du contrôle (justifier votre réponse).

il s'agit uniquement du noeud B_1 , c'est le seul noeud à partir duquel il existe un chemin vers B_5 dans lequel B_5 postdomine tous les noeuds mais B_5 ne postdomine pas B_1

40 Analyse data-flow

Cet exercice a pour but de réaliser la transformation de *strength reduction* en la formulant comme une analyse data-flow. Pour cela on procède en deux étapes, on détecte les variables d'induction primaires puis on repère les opérations candidates (ou variables d'inductions générales). On va d'abord s'intéresser au premier problème : la détection des variables d'induction primaire.

On supposera dans un premier temps que l'on travaille sur une région où il n'y a qu'une seule boucle, comme par exemple le graphe de contrôle de flot de la figure 14. Il n'y a donc pas d'ambiguïté lorsque l'on parle de constante de région : il s'agit d'une variable ou d'une valeur constante dans tous les blocs de la boucle du programme. D'autre part, comme on ne recherche que les variables d'induction *primaires*, on supposera que la seule forme de mise à jour d'une variable d'induction primaire autorisée est : $iv = iv + c$ où iv est une variable d'induction et c une constante de région (on suppose que l'on connaît l'ensemble \mathcal{C} des constantes de région).

Question 40-1

Formulez une analyse data-flow dont la solution donne les variables d'induction primaires de la boucle de la région considérée. On demande ici simplement de définir informellement (mais précisément) les ensembles et les équations que doivent vérifier ces ensembles.

L'idée importante ici c'est que les ensembles que l'on recherche ne sont pas simplement des ensemble de variables, mais des ensembles de couple (variable,incrément). En effet, il faut être capable de détecter les cas où deux incréments différents sont appliqués sur une variables (comme sur m par exemple). On définit le prédicat $IVDQ(b, v)$ vrai si la mise à jour de la variable v dans le bloc b est une opération valide de mise à jour d'une variable d'induction (Induction Variable Definition Q). On définit ensuite les ensembles $IVDGen$, $IVDKill$, $IVDin$, $IVDout$ et $IVDKillAll$ de la manière suivante :

- $IVDGen(b)$ est l'ensemble des couples (variables,constante) dont la mise à jour de la variable par le bloc b est compatible avec la mise à jour d'une variable d'induction :

$$IVDGen(b) = \{(n, c) \mid IVDQ(b, n) \&\& \text{ la variable } n \text{ n'est pas tuée avant la fin du bloc } b\}$$

- $IVDKill(b)$ est l'ensemble de variable d'induction tuées par le bloc b (le bloc b les assigne avec une expression qui ne définit pas une variable d'induction).

$$IVDKill(b) = \{(n, *) \mid \text{ la variable } n \text{ est tuée dans le bloc } b \text{ sans définir une variable d'induction}\}$$

$IVDKill(b)$ et $IVDGen(b)$ peut être calculées sans itération.

- $IVDin(b)$ est l'ensemble des variables d'inductions vivantes à l'entrée du bloc b .
- $IVDout(b)$ est l'ensemble des variables d'inductions vivantes à la sortie du bloc b .
- $IVDKillAll(b)$ est l'ensemble des variables d'inductions tuées par un quelconque des blocs de la boucle.

Ces ensembles doivent vérifier les équations suivantes :

$$\begin{aligned} IVDKillAll(b) &= IVDKillAll(b) \cup_{d \in \text{pred_in_loop}(b)} IVDKill(d) \\ IVDin(b) &= \bigcap_{d \in \text{pred_in_loop}(b)} IVDout(d) \\ IVDout(b) &= (IVDGen(b) \cup IVDin(b)) - IVDKillAll(b) \end{aligned}$$

Question 40-2

Itérez ces équations sur l'exemple de la figure 14

On calcule d'abord $IVDKill$ et $IVDGen$, puis à chaque itération $IVDin$ puis $IVDout$ dans l'ordre $B_2 \dots B_5$.

$$\begin{array}{ll} IVDKill(B_3) &= \{(t1, *), (j, *), (k, *)\} & IVDGen(B_3) &= \emptyset \\ IVDKill(B_4) &= \emptyset & IVDGen(B_4) &= \{(i, 1), (m, 1)\} \\ IVDKill(B_5) &= \{(m, *)\} & IVDGen(B_5) &= \{(i, 1)\} \end{array}$$

en notant $killall = \{(m, *), (t1, *), (j, *), (k, *)\}$:

set	iteration 0	1	2	3	4 arret
$IVDin(B_2)$	\emptyset	$\{(i, 1)\}$	$\{(i, 1)\}$	$\{(i, 1)\}$	$\{(i, 1)\}$
$IVDout(B_2)$	\emptyset	$\{(i, 1)\}$	$\{(i, 1)\}$	$\{(i, 1)\}$	$\{(i, 1)\}$
$IVDKillAll(B_2)$	\emptyset	$\{(m, *)\}$	$killall$	$killall$	$killall$
$IVDin(B_3)$	\emptyset	\emptyset	$\{(i, 1)\}$	$\{(i, 1)\}$	$\{(i, 1)\}$
$IVDout(B_3)$	\emptyset	\emptyset	$\{(i, 1)\}$	$\{(i, 1)\}$	$\{(i, 1)\}$
$IVDKillAll(B_3)$	$\{(t1, *), (j, *), (k, *)\}$	$\{(t1, *), (j, *), (k, *)\}$	$killall$	$killall$	$killall$
$IVDin(B_4)$	\emptyset	\emptyset	\emptyset	$\{(i, 1)\}$	$\{(i, 1)\}$
$IVDout(B_4)$	$\{(i, 1), (m, *)\}$	$\{(i, 1), (m, *)\}$	$\{(i, 1), (m, *)\}$	$\{(i, 1)\}$	$\{(i, 1)\}$
$IVDKillAll(B_4)$	\emptyset	$\{(t1, *), (j, *), (k, *)\}$	$\{(t1, *), (j, *), (k, *)\}$	$killall$	$killall$
$IVDin(B_5)$	\emptyset	\emptyset	\emptyset	$\{(i, 1)\}$	$\{(i, 1)\}$
$IVDout(B_5)$	$\{(i, 1)\}$	$\{(i, 1)\}$	$\{(i, 1)\}$	$\{(i, 1)\}$	$\{(i, 1)\}$
$IVDKillAll(B_5)$	$\{(m, *)\}$	$killall$	$killall$	$killall$	$killall$

Notons que cette correction n'est pas très élégante mais celle proposée dans le Muchnick me paraît fausse.

Question 40-3

On recherche maintenant les opérations candidates à une réduction de la forme suivante : $x \leftarrow iv \times c_1 \pm c_2$, ou iv est une variable d'induction primaire et c_1 et c_2 sont des constantes de région. Doit-on résoudre un nouveau problème data-flow pour identifier ces opérations ?

Non, il suffit de faire une passe linéaire pour repérer ces opérations, ceci dit la réponse oui peut convenir aussi, car ces opérations candidates donnent lieu à de nouvelles variables d'induction (dites générales), il faut donc itérer le processus...

Question 40-4

Indiquer comment généraliser la méthode pour traiter les boucles imbriquées.

L'idée c'est qu'il faut rajouter un argument aux fonctions IVD* qui est l'ensemble des blocs de la boucle considérées (on paramètre la fonction par une région du graphe qui constitue une boucle, que l'on peut identifier statiquement avant l'analyse