Cours de Compilation

Rédaction : Laure Danthony, MIM00

Cours : Yves Robert, septembre - décembre 2001

Table des matières

1 Introduction **5**
- 1.1 Généralités . . . 5
- 1.2 Compilation VS interprétation . . . 5
- 1.3 Grammaires . . . 6
- 1.4 Algorithme de fermeture transitive . . . 7

2 Analyse lexicale **9**
- 2.1 Introduction . . . 9
- 2.2 Expressions régulières . . . 10
- 2.3 Méthode des dotted items . . . 11
- 2.4 Complexité . . . 12
- 2.5 Complications . . . 13

3 Analyse syntaxique **15**
- 3.1 Exemples . . . 15
- 3.2 Parsing LL(1) . . . 16
  - 3.2.1 L'ensemble FIRST . . . 16
  - 3.2.2 L'ensemble FOLLOW . . . 18
  - 3.2.3 Problèmes . . . 18
  - 3.2.4 Automates à pile LL . . . 19
- 3.3 Parsing LR . . . 20
  - 3.3.1 Analyse de LR(0) . . . 20
  - 3.3.2 Analyse SLR(1) . . . 25
  - 3.3.3 Analyse LR(1) . . . 26

4 Attributs **29**
- 4.1 Exemples et notions de base . . . 29
- 4.2 Un peu de théorie . . . 32
- 4.3 Liens entre les grammaires S- et L-attribuées . . . 32

5 Contrôle statique **35**
- 5.1 Graphe de flot de contrôle . . . 35
  - 5.1.1 Expressions . . . 35
  - 5.1.2 If statement . . . 35
  - 5.1.3 Case statement . . . 36
  - 5.1.4 Boucles For . . . 37
- 5.2 Contrôle de types . . . 37
  - 5.2.1 Scopes . . . 37

5.2.2 Vérification de types 38

6 Procédures et objets **41**
6.1 Notion d'Activation record 41
6.2 Appels imbriqués 42
6.3 Détails d'un appel 43
6.3.1 Qu'est-ce-qu'on appelle ? 43
6.3.2 Comment on fait ? 43
6.4 Quelques remarques sur les objets 43
6.4.1 Héritage 44
6.4.2 Redéfinition des méthodes 44
6.4.3 Polymorphisme 44
6.5 Langages fonctionnels 44

7 Génération de code **47**
7.1 Introduction 47
7.2 Génération avec des machines à base de registres 47
7.3 Basic blocks 50
7.3.1 De l'AST au DDG 50
7.3.2 Du DDG au code : les échelles 51
7.4 Sélection des instructions 54
7.4.1 Bottom-up pattern-matching 55
7.4.2 Instruction selection by dynamic programming 55
7.4.3 BURS : Bottom-up Rewriting system 56

8 Registres **63**
8.1 Liveness Analysis (vivacité) 63
8.1.1 Exemple 63
8.1.2 Equation data-flow 64
8.1.3 Static vs dynamic liveness 65
8.1.4 Graphe d'interférence 66
8.1.5 Traitement des move 66
8.2 Allocation de registres 66
8.2.1 Coloriage par simplification 67
8.2.2 Traitement des move 68
8.2.3 Noeuds précolorés 70
8.3 Traitement d'un exemple complet 70

9 Analyse data flow **77**
9.1 Reaching definitions 77
9.2 Available expressions 78
9.3 Optimisation avec l'analyse data-flow 79
9.4 Accélération 80
9.4.1 Au niveau des basic blocks 80
9.4.2 Ordre des nœuds 80
9.4.3 Value numbering 81
9.5 Alias analysis 81

10 Références **83**

Chapitre 1

Introduction

1.1 Généralités

Pour la production d'un exécutable à partir d'un code source, on distingue généralement les étapes suivantes :
texte source → front-end → représentation sémantique → back-end → exécutable.

En général, on considère que les trois étapes intermédiaires se font à l'intérieur du compilateur :

- *front-end* : analyse lexicale (qui mène à la production de lexèmes / tokens), puis l'analyse syntaxique qui aboutit à la construction de l'arbre syntaxique (AST).
- Ensuite, la sémantique annote cet AST (types, identificateurs) et on aboutit à la représentation sémantique.
- *back-end* : optimisation du code intermédiaire (sous expressions communes : on ne veut faire qu'une seule évaluation, *etc.*On ne fait que des trucs indépendants de la machine). On produit ensuite le code assembleur / le code machine. Enfin on ajoute les bibliothèques, on résout les conflits.
- Le produit est l'exécutable.

1.2 Compilation VS interprétation

En général on distingue deux types de langages : les langages compilés et les langages interprétés. Grosso-modo, on considère que le travail de l'interpréteur se réduit au front-end (pas de back-end ou presque) :

- Compilation : le compilateur produit un programme cible à partir du code source. C'est ensuite ce programme qui va fournir à l'utilisateur un résultat à partir de l'input.
- Interprétation : A partir du source et de l'input, on donne directement le résultat.

Le compilateur est plus rapide, mais il est plus difficile à écrire. L'interpréteur est plus facile à débogger (il traite les instructions au vol donc on peut "tracer" l'exécution, et il est plus sécuritaire.

Pour schématiser d'avantage :

	Compilateur	Interpréteur
traitement du source	énorme	limité
forme intermédiaire	exécutable binaire	structure de données spécifique
mécanisme d'interprétation	CPU	soft
vitesse	rapide	lente

1.3 Grammaires

On étudie ici succinctement les grammaires hors contexte. Pour plus de détails, on pourra se reporter au cours de Langages Formels.

DÉFINITION 1.3.1
Une **grammaire hors contexte** est un quadruplet $G = (V_N, V_T, S, P)$ où :
- V_N est un alphabet, dit alphabet des non-terminaux ;
- V_T est un alphabet, dit alphabet des terminaux, $V_T \cap V_N = \emptyset$;
- $S \in V_N$ est dit élément de départ (ou axiome de G) ;
- P est un ensemble de productions, c'est à dire de règles du type $N \to \alpha$ avec $N \in V_N$ et $\alpha \in (V_N \cup V_T)^*$ un mot (ou "chaîne").

DÉFINITION 1.3.2 (DÉRIVATION)
On dit que la chaîne β est **directement dérivable de** α si $\exists \gamma, S_1, S_2$ chaînes, $\exists N \in V_N$ tels que $\alpha = S_1 N S_2$ et $\beta = S_1 \gamma S_2$ où $N \to \gamma$ est une production de P.

DÉFINITION 1.3.3 (PHRASE)
Une **phrase** est une chaîne α que l'on peut obtenir à partir de l'axiome de G par dérivations successives : $S \to^* \alpha$.
Une **production terminale** est une phrase qui est dans V_T^*.

DÉFINITION 1.3.4
Le **langage engendré par la grammaire** G, noté $\mathcal{L}(G)$ est l'ensemble des productions terminales.

EXEMPLE 1.3.1 *Le langage des expressions totalement parenthésées (avec* $1, +, *$ *est généré par la grammaire suivante :*
- $S = exp$
- $V_N = \{exp, op\}$
- $V_T = \{1, (,), +, *\}$
- *P est constitué des productions suivantes :*

  1. $exp \to' (' \ exp \ op \ exp \ ')'$
  2. $exp \to 1$
  3. $op \to +$
  4. $op \to *$

EXERCICE 1.3.1 *Dériver* $(1 * (1 + 1))$.
On peut vérifier : $S \to_1 \to_2 \to_4 \to_1 \to_2 \to_3 \to_4 (1*(1+1))$ *Donc* $(1*(1+1)) \in \mathcal{L}(G)$

DÉFINITION 1.3.5 (RÉCURSIF)
$N \in V_N$ est dit **récursif à gauche** si à partir de N on peut dériver une phrase commençant par N.

DÉFINITION 1.3.6 (ANNULABLE)
$N \in V_N$ est dit **non terminal annulable** si à partir de N on peut dériver une ε la chaîne vide.

DÉFINITION 1.3.7 (INUTILE)
$N \in V_N$ est dit **non terminal inutile** si à partir de N on ne peut pas dériver une chaîne de terminaux.

DÉFINITION 1.3.8 (GRAMMAIRE AMBIGUË)
Une grammaire est dite **ambiguë** si on peut obtenir deux arbres de dérivation différents avec des feuilles dans le même ordre.

REMARQUE : On utilise souvent la notation A, B, C pour les non terminaux, x, y, z pour les terminaux, les chaînes sont notées α, β, ε pour la chaîne vide. Enfin on dénote a, b, c des terminaux égaux à eux-mêmes (par opposition aux terminaux "inconnus" x, y, z).

EXEMPLE 1.3.2 *Considérons la grammaire définie par :*

1. $S \to A|B|C$ *(abus de langage !)*
2. $A \to B|\varepsilon$
3. $B \to x|Cy$
4. $C \to BCS$

Alors : cette grammaire est ambiguë car $S \to B \to x$ *et* $S \to A \to B \to x$. S *et* A *sont annulables* ($S \to A \to \varepsilon$), C *est récursif à gauche* ($C \to BCS \to CyCS$) *et* B *aussi. Enfin* C *est inutile. Le langage est* $\mathcal{L}(G) = \{\varepsilon, x\}$.

1.4 Algorithme de fermeture transitive

Il peut être intéressant de calculer le graphe des appels des procédures d'un programme donné, notamment pour savoir quelle procédure appelle quelle autre, *etc.*Pour savoir si un programme est suseptible d'en appeler un autre, on souhaite pouvoir avoir la fermeture transitive d'un tel graphe.

L'algorithme de Warshall-Floyd fait cela en $O(n^3)$ mais en un espace $O(n^2)$. Or ici les graphes sont très creux donc on utilise un algo incrémental ($O(n^5)$ au pire mais dans les cas pratiques cela ira plus vite :

```
DEBUT
changé := true;
while changé
   changé:=false;
   pour tout sommet s1 du graphe
      pour tout successeur s2 de s1
         pour chaque successeur s3 de s2
              si il n'existe pas s1->s3, la mettre et changé:=true
              sinon rien
```

```
        finpour
      finpour
   finpour
FIN
```

Chapitre 2

Analyse lexicale

2.1 Introduction

Le but de l'analyse lexicale est d'envoyer des expressions régulières à l'analyse syntaxique qui les transformera en AST (arbre).

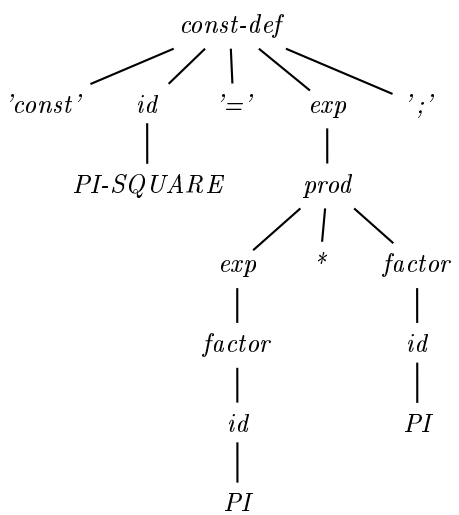
EXEMPLE 2.1.1 *On considère les définitions de constantes "à la Pascal" :*

```
CONST PI=3.14;
CONST PI_SQUARE=PI*PI;
```

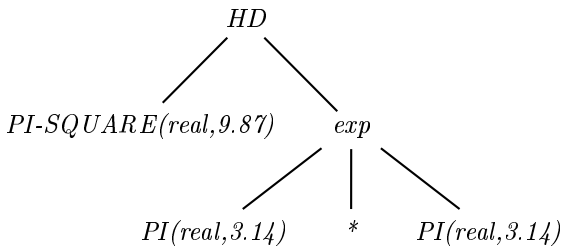
La grammaire correspondante est constituée des productions :

- *const-def* $\rightarrow$ *'CONST' id '=' exp ';'*
- *exp* $\rightarrow$ *prod* | *factor*
- *prod* $\rightarrow$ *exp* $*$ *factor*
- *factor* $\rightarrow$ *number* | *id*

Le but est d'obtenir l'arbre suivant :



Ensuite, on annotera cet arbre et on le compactera pour obtenir l'arbre suivant :



Cette opération dépasse largement le cadre de l'analyse lexicale.

Le but de l'analyse est de trouver des "tokens" (unités de sens), et des classes pour ces tokens, par exemple :

```
struct const-def{
   id  = *CD-id;
   exp = *CD-exp;
}
```

Dans la suite, on considère que tout le source rentre dans un buffer, *i.e.* que l'on a pas besoin de s'embêter pour le parcours du texte, notamment pour le backtracking. Dans la réalité on utilise des flux.

2.2 Expressions régulières

DÉFINITION 2.2.1 (TOKEN)
Un token est le plus petit ensemble de caractères qui ait un sens et dont le sens est changé si on ajoute un caractère blanc au milieu.

REMARQUE : Lorsque l'on va essayer de détecter des tokens, on va essayer de matcher une plus grande chaîne possible. Mais attention, l'ordre d'écriture de la grammaire compte (on essaie les règles dans cet ordre). Par exemple, il faut écrire dans l'ordre :

magic_symbol → xyzzyx
identifier → [a-z]+

si on veut différencier magic_symbol d'un identificateur ordinaire.

Les notations sont à peu près standardisées :
- `x` désigne le caractère x.
- `[a-z]` désigne 1 caractère entre a et z.
- `.` désigne n'importe quel caractère.
- `R1R2` est utilisé pour la concaténation.
- `R1|R2` : union.
- `R*` : fermeture de Kleene.
- `R+` : au moins un R.

Lorsqu'il y a ambiguïté, on utilise des parenthèses. La fermeture de Kleene est prioritaire sur le reste.

EXEMPLE 2.2.1
- *Les chaînes de* 0 *et de* 1 *qui ont deux* 0 *consécutifs :* $(0|1)^*\ 00\ (0|1)^*$
- *Le complémentaire du précédent :* $1^*(01^+)^*(0|\varepsilon)$.
- *Nombre pair de* 1 *:* $\left((10^*1)^*|0^*\right)^*$.

2.3 Méthode des dotted items

Dans cette section on introduit une méthode non canonique pour l'analyse lexicale. On obtiendra les automates habituels comme conséquence. Le "point" noté $\bullet$ sera utilisé comme un curseur. On va créer des productions nouvelles à la grammaire, appelées *items*, qui contiendront ce point.

Le "point" sépare dans $\tau \to \alpha\beta$ ce qu'on a déjà "matché" de ce qu'on a pas encore fait, ce que l'on note $\tau \to \alpha \bullet \beta$. Un item non basique est un item où le "point" est juste avant une sous-expression régulière de type opérateur de répétition ou expression parenthésée. Il y a deux catégories d'items basiques, suivant que le point est tout à la fin ou non :

DÉFINITION 2.3.1 (BASIC)
Les items basiques sont :
- $\alpha \bullet exprbasic$ appelé communément "shift".
- $\alpha\bullet$ (*i.e.* le point à la fin) appelé "reduce".

Il faut gérer les items non basiques : on fait des transformation de productions, résumées ci-dessous :

- On remplace $\tau \to \alpha \bullet (R)^*\beta$ par : $\begin{cases} \tau \to \alpha(R^*) \bullet \beta & \text{((i) plus de R)} \\ \tau \to \alpha(\bullet R)^*\beta & \text{((ii) encore un R)} \end{cases}$

  Détaillons : on part de $\tau \to \alpha \bullet (R)^*\beta$; soit R n'est pas présent, et on a le premier cas (i). Soit R est présent, et on passe à $\tau \to \alpha(R\bullet)^*\beta$, qu'on remplace par (i) si c'était la dernière occurrence de R, et par (ii) sinon.
- On remplace $\tau \to \alpha(R\bullet)^*\beta$ par $\begin{cases} \tau \to \alpha(\bullet R)^*\beta \\ \tau \to \alpha(R^*) \bullet \beta \end{cases}$
- On remplace $\tau \to \alpha \bullet (R)^+\beta$ par $\tau \to \alpha(\bullet R)^+\beta$.
- On remplace $\tau \to \alpha(R\bullet)^+\beta$ par $\begin{cases} \tau \to \alpha(\bullet R)^*\beta \\ \tau \to \alpha(R^*) \bullet \beta \end{cases}$

REMARQUE : On a géré ce qu'on appelle des ε *move* (sans rien consommer d'input) pour se ramener, en traitant tous les cas possibles, à des items basiques.

EXEMPLE 2.3.1 *On dispose de la grammaire suivante :* $\begin{cases} int \to [0-9]+ \\ real \to [0-9]* \ '.'[0-9]+ \end{cases}$
On va essayer de reconnaître le réel 3.1. *Pour cela on essaie de matcher en parallèle avec un int et un real :*

- *Etape 0 :* $S_0 = \begin{cases} int \to \bullet([0-9])^+ \\ real \to \bullet([0-9])^* \ '.'([0-9])^+ \end{cases}$

  Comme on a des items non basiques, il faut effectuer les ϵ move *pour obtenir* $S_0 = \begin{cases} int \to (\bullet[0-9])^+ \\ real \to (\bullet[0-9])^* \ '.'([0-9])^+ \\ real \to ([0-9])^* \ \bullet' .'([0-9])^+ \end{cases}$
- *Etape 1 : On traite le* 3 *en entrée; la dernière dérivation attendait un point, elle est abandonnée. Restent :* $S_1 = \begin{cases} int \to ([0-9]\bullet)^+ \\ real \to ([0-9]\bullet)^* \ '.'([0-9])^+ \end{cases}$

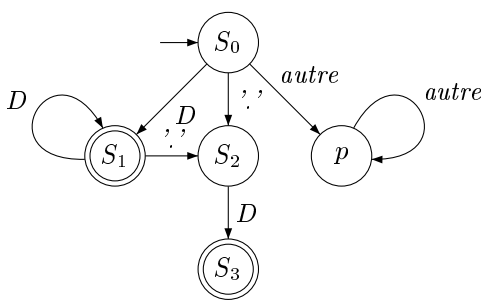
  qui se développe avec les ε move *en :*

$$S_1 = \begin{cases} int \to (\bullet[0-9])^+ \\ int \to ([0-9])^+\bullet & <<<< \textit{reconnu} \\ real \to (\bullet[0-9])^* \ '.'([0-9])^+ \\ real \to ([0-9])^* \ \ \bullet' .'([0-9])^+ \end{cases}$$

- *Etape 2 : On traite le '.' en entrée :* $S_2 = \Big\{ real \to ([0-9])^* \ '.' \bullet ([0-9])^+$
  qui se développe en : $S_2 = \Big\{ real \to ([0-9])^* \ '.'(\bullet[0-9])^+$
- *Etape 3 : On traite le* 1 *en entrée :* $S_3 = \Big\{ real \to ([0-9])^* \ '.'([0-9]\bullet)^+$
  qui se développe en :
  $$S_3 = \begin{cases} real \to ([0-9])^* \ '.'(\bullet[0-9])^+ \\ real \to ([0-9])^* \ '.'([0-9])^+\bullet & <<<< \textit{reconnu} \end{cases}$$

Lorsque le point est à la fin, on accepte jusqu'à ce que éventuellement il y ait quelque chose de plus long (éventuellement ici on pourrait avoir 3.14*). On garde toujours en mémoire la chaîne la plus longue reconnue jusqu'ici.*

Si on prend pour chaque état les ensembles d'items précédents (après leur avoir fait subir une fermeture transitive sur les expressions non basiques), on obtient, en notant D *pour* $[0-9]$ *et* I *pour int,* R *pour real :*



les états finaux sont les états qui contiennent une production avec un point final. Les flèches manquantes vont au puits p*.*

D'où la table des transitions :

state/char	D	'.'	*autre*	*token-class*
S_0	S_1	S_2	–	–
S_1	S_1	S_2	–	*int*
S_2	S_3	–	–	–
S_3	S_3	–	–	*real*

REMARQUE : Dans la réalité, on compacte la table.

2.4 Complexité

Tout d'abord, remarquons que l'analyse syntaxique est la seule étape de la compilation où l'input est entièrement scanné. Ensuite, la taille de ce qu'il y a à analyser est considérablement réduite.

Est-ce que cette étape s'effectue en $O(n)$? Et bien non, comme le montre l'exemple suivant :

EXEMPLE 2.4.1 *Sur la grammaire :* $\alpha \rightarrow' a'$ *et* $\beta \rightarrow ('a')^*b$ *et sur l'input* a^n*, lorsque l'on a lu le dernier a, on accepte le premier a et on revient en arrière de* $n-1$ *positions, puis de* $n-2$ *positions après avoir accepté le second a, ainsi de suite !*

D'où une complexité en temps au pire $O(n^2)$.

De même si la taille de la table peut être réduite, au pire l'automate construit a un nombre exponentiel d'états :

EXEMPLE 2.4.2 *Considérons la production* $\alpha \rightarrow .^*a....b$ *(avec k "points" séparant a et b), l'automate aura* 2^{k+1} *états, car il faudra se souvenir des positions des a dans les* $k+1$ *dernières places, soit* 2^{k+1} *possibilités, ce qui exige autant d'états pour les différencier.*

2.5 Complications

Nous allons voir dans cette section quelques exemples de complications.

EXEMPLE 2.5.1 *Selon le contexte,* `(T *)` *peut être soit mis pour appeler* `int T` *et on a oublié une opérande après* `*`*, ou alors on a créé un alias* `T` *pour* `int` *(autrement dit un* typedef*) et on veut caster ce qui suit vers un* `int`*.*

On peut aussi avoir des soucis avec les définitions des macros (qu'il faut étendre (!) lorsque on les rencontre ; il faut tenir compte des mots-clefs du langage qui ne peuvent pas être des identificateurs comme les autres ; les listes de définitions (structures, procédures, peuvent aussi poser problème.

REMARQUE : Un mot sur la table des symboles : on voudrait pouvoir indexer un tableau par des chaînes : on utilise pour cela des tables de hachages (tableaux de listes où on met le dernier rencontré devant).

EXEMPLE 2.5.2 *Avec des définitions de macro, à un moment donné de l'analyse on peut se trouver dans la situation qui suit :*

buffer	*contenu*
input.c	`...#include mac.h...`
mac.h	`...is_capital(text[i])...`
macro is_capital	`('A'<=(ch)) && ((ch) <= 'Z')`
parameter	`text[i]`

L'analyse doit en particulier comprendre l'expansion du paramètre, et ainsi en cascade le nombre de buffers à gérer en même temps peut être dramatique.

En résumé, l'analyse lexicale n'est pas aussi simple qu'on ne le croit !

Chapitre 3

Analyse syntaxique

3.1 Exemples

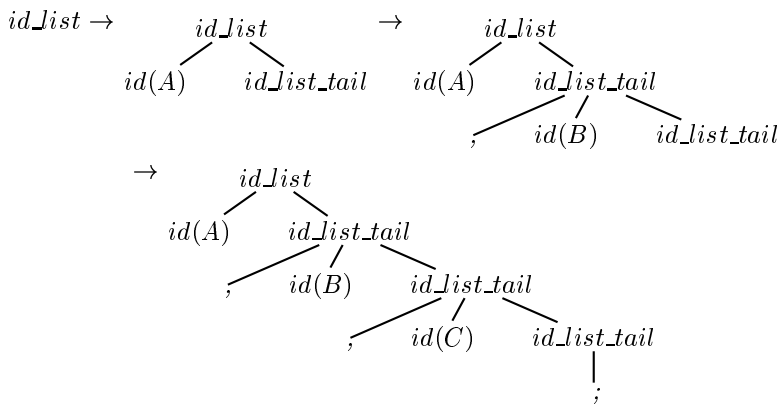
Le but de l'analyse syntaxique est de mettre les Tokens renvoyés par l'analyse lexicale sous forme d'AST (Abstract syntax Tree), qui a une structure porteuse de sens.

EXEMPLE 3.1.1 *On essaie de parser* `A,B,C;`*, on pense pour cela à deux grammaires différentes, donc dont le parsing va être différent :*

- *(A)* $\begin{cases} id_list \rightarrow id_list_prefix\ ; \\ id_list_prefix \rightarrow id \mid id_list_prefix, id \end{cases}$
- *(B)* $\begin{cases} id_list \rightarrow id\ id_list_tail \\ id_list_tail \rightarrow\ ,\ id\ id_list_tail \mid ; \end{cases}$

Différences : A est récursive à gauche, pas B. La grammaire A doit être parsée "bottom up" (elle est LR), et B peut être parsée "top down"(elle est LL) ou "bottom up".

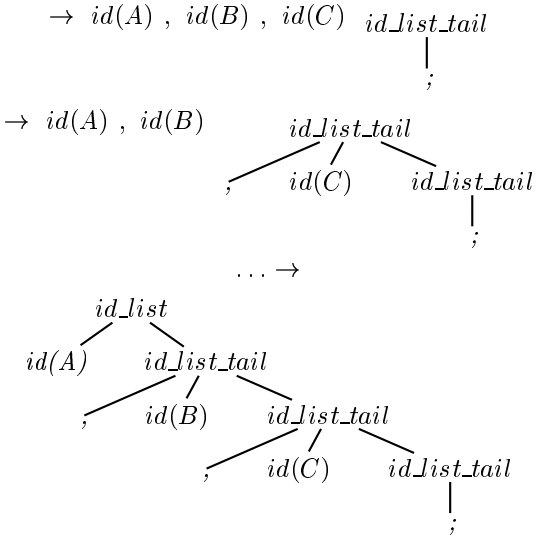
Mais que veulent bien dire ces différentes sortes de parsing ? Essayons de parser B (sur `A,B,C;`*) "top down". On fait les étapes successives :*



Essayons de parser B maintenant "bottom up" :

$id(A) \rightarrow id(A)\ ,\ \rightarrow id(A), id(B) \rightarrow\ id(A)\ ,\ id(B),\ \rightarrow\ id(A)\ ,\ id(B)\ ,\ id(C)\ ;$

$\rightarrow\ id(A)\ ,\ id(B)\ ,\ id(C)$



En exercice, essayer de parser A "bottom-up" : c'est possible car à aucun moment on n'a le choix. Par contre il y a un problème car on a deux choix possibles à chaque fois si on essaie de parser A "top down".

3.2 Parsing LL(1)

Ici on va calculer des sous-ensembles de lexèmes qui nous aideront en pratique à parser une expression.

Dans toute la suite, on traite l'exemple de la grammaire suivante :

EXEMPLE 3.2.1

$$G = \begin{cases} input \rightarrow\ exp\ EOF \\ exp \rightarrow\ term\ rest_exp \\ term \rightarrow\ id \mid parent_exp \\ parent_exp \rightarrow\ '('\ exp\ ')' \\ rest_exp \rightarrow\ '+'\ exp \mid \varepsilon \end{cases}$$

3.2.1 L'ensemble FIRST

DÉFINITION 3.2.1 (FIRST)
$FIRST(N)$ contient l'ensemble des premiers tokens produits par au moins une dérivation de N.

Le but de cette section est de trouver FIRST pour :
- tous les terminaux,
- tous les non terminaux,
- les alternatives (*i.e.* ce qu'il y a à droite de la flèche) pour les non terminaux,

– les fins d'alternatives.

ALGO :

INIT
– $\forall T$ terminal, $FIRST(T) = T$
– $\forall N$ non terminal, $FIRST(N) = \emptyset$
– $\forall \alpha$ non vide alternative ou fin d'alternative, $FIRST(\alpha) = \emptyset$
– $\forall \alpha$ alternative vide, $FIRST(\alpha) = \{\varepsilon\}$

CORPS - Examen des règles :

1. $N \to \alpha$: $FIRST(N)$ doit contenir $FIRST(\alpha)$, y compris éventuellement ε.
2. $\forall \alpha$ alternative ou fin d'alternative s'écrivant $\alpha = A\beta$: $FIRST(\alpha)$ doit contenir $FIRST(A)$, sauf ε si il y est.
3. $\forall \alpha$ alternative ou fin d'alternative s'écrivant $\alpha = A\beta$: si $\varepsilon \in FIRST(A)$ alors $FIRST(\alpha)$ doit contenir $FIRST(\beta)$, sauf ε si il y est.

On fait donc une fermeture transitive du tout, et c'est dans la poche ! Montrons le résultat sur l'exemple :

mots	init	après_algo
input	$\emptyset$	$\{id, '('\}$
exp EOF	$\emptyset$	$\{id, '('\}$
EOF	$\{EOF\}$	$\{EOF\}$
exp	$\emptyset$	$\{id, '('\}$
term rest_exp	$\emptyset$	$\{id, '('\}$
rest_exp	$\emptyset$	$\{'+', \varepsilon\}$
term	$\emptyset$	$\{id, '('\}$
id	$\{id\}$	$\{id\}$
| parent_exp	$\emptyset$	$\{'+', \varepsilon\}$
'(' exp ')'	$\{'('\}$	$\{'('\}$
exp ')'	$\emptyset$	$\{id, '('\}$
'+' exp	$\emptyset$	$\{+\}$
| ε	$\{\varepsilon\}$	$\{\varepsilon\}$
'+'	$\{+\}$	$\{+\}$
'('	$\{'('\}$	$\{'('\}$
')'	$\{')'\}$	$\{')'\}$

REMARQUE : Ce n'est pas obligatoire de stocker $FIRST$ pour des alternatives. Ceci dit c'est pratique car cela évite de recalculer la même fermeture transitive lors de l'analyse proprement dite.

On obtient alors quelque chose du type :

```
void input(void)
{
   switch(Token.class){
      case id : case'(': exp();token(EOF);break;
      default : error();
   }
}
```

```
void term(void)
{
   switch(Token.class){
      case id : token(id);break;
      case '(': par_exp();break;
      default : error();
   }
}
```

Mais il y a un problème lorsque $FIRST$ contient ε (on ne peut pas matcher ε) :

```
void rest_exp(void)
{
   switch(Token.class){
      case '+' : token(+);exp();break;
      case EOF : case ')' : break;
      default  : error();
   }
}
```

Il faut alors calculer l'ensemble des deuxièmes tokens (EOF et $')'$ appartiennent à $FOLLOW(rest_exp)$). D'où le paragraphe suivant.

3.2.2 L'ensemble FOLLOW

DÉFINITION 3.2.2
$FOLLOW(N)$ contient l'ensemble des tokens qui suivent le premier token pouvant dériver N.

Le but du jeu est de calculer l'ensemble $FOLLOW$ pour chaque non terminal :

ALGO :

INIT Tous les $FOLLOW$ sont initialisés à $\emptyset$. On utilise les ensembles $FIRST$ que l'on vient de calculer .

CORPS - Examen des règles :

1. $M \to \alpha N \beta$: $FOLLOW(N)$ doit contenir $FIRST(\beta)$.
2. $M \to \alpha N \beta$ et $\varepsilon \in FIRST(\beta)$: $FOLLOW(N)$ doit contenir $FOLLOW(M)$.

On obtient alors :

mots	FIRST	FOLLOW
input	$\{id, '('\}$	$\emptyset$
exp	$\{id, '('\}$	$\{EOF, ')'\}$
term	$\{id, '('\}$	$\{'+', EOF, ')'\}$
parent_exp	$\{'('\}$	$\{'+', EOF, ')'\}$
rest_exp	$\{'+', \varepsilon\}$	$\{EOF, ')'\}$

3.2.3 Problèmes

Les problèmes que l'on peut rencontrer sont les suivants :

- Conflit FIRST/FIRST Par exemple, si on ajoute à la grammaire précédente les deux règles suivantes : $term \rightarrow id \mid tab \mid parent_exp$ et $tab \rightarrow id\ '[\ 'exp\ ']'$, lorsque l'on voit id, on ne sait pas si on doit accepter ou continuer en attendant $'['$.

  Pour nous, LL sera une grammaire telle que l'analyse avec FIRST et FOLLOW ne conduise à aucun conflit (pour plus de détails, voir le cours de langages).

- Conflit FIRST/ FOLLOW Considérons la grammaire suivante :
  $\begin{cases} S \rightarrow A\ 'a'\ 'b' \\ A \rightarrow'\ a'\ \mid \varepsilon \end{cases}$

  Cette grammaire reconnaît les deux seuls mots ab et aab. Que donnent les deux ensembles FIRST et FOLLOW ?
  – $FIRST(S) = \{'a'\}$, $FIRST(A) = \{'a',\ \varepsilon\}$,
  – $FOLLOW(S) = \emptyset$, $FOLLOW(A) = \{'a'\}$.

  Lorsque l'on est dans la situation $A \bullet\ 'a'$, est-ce qu'on réduit en utilisant la deuxième règle, ou est-ce qu'on shifte en appliquant la règle 1 ?

Ce qui nous mène à la définition suivante :

Une grammaire $LL(1)$ est une grammaire telle que lors du traitement avec FIRST/FOLLOW, elle ne conduit pas :
– **à un conflit FIRST/FIRST** : si N a plusieurs alternatives possibles, les ensembles FIRST de ces diverses alternatives doivent être disjointes.
– **à un conflit FIRST/FOLLOW** : si N est annulable, il faut que les ensembles $FOLLOW(N)$ et $FIRST(\alpha)$, pour α alternative de N, soient d'intersection nulle.

Comment rendre une grammaire LL(1) ?

Il y a trois méthodes pour rendre une grammaire LL :
- **Factorisation à gauche**. Elle va éliminer les conflits FIRST/FIRST : on remplace la règle $term \rightarrow id \mid id\ '[\ exp\ ']' \mid \ldots$ par les deux règles $term \rightarrow id\ rest_id \mid \ldots$ et $rest_d \rightarrow\ '[\ exp\ ']' \mid \varepsilon$.
- **Substitution**. En général, on a besoin de substituer avant de factoriser à gauche. On remplace les deux règles suivantes : $A \rightarrow a \mid Bc \mid \varepsilon$ et $S \rightarrow pAq$ par $S \rightarrow\ paq \mid pBcq \mid q$.
- **Elimination de la récursivité à gauche**. La récursivité à gauche peut être directe ($N \rightarrow N$), indirecte ($N \rightarrow A$, $A \rightarrow B$, $B \rightarrow N$), ou cachée ($N \rightarrow \alpha\ N, \alpha \rightarrow \varepsilon \mid \ldots$). Sur l'exemple $N \rightarrow N\alpha \mid \beta$ (directe), on remplace par $N \rightarrow \beta\ N'$ et $N' \rightarrow \varepsilon \mid \alpha N'$.

3.2.4 Automates à pile LL

Lorsque l'on a parsé à l'aide de FIRST et FOLLOW, on a en fait obtenu la table suivante :

next state / top of stack	id	+	(	)	EOF
input	exp		exp EOF		
exp	term rest_exp		term rest_exp		
term	id		par_exp		
par_exp			(exp)		
rest_exp		+exp		ε	ε

Cela fait penser à un automate. Ceci dit, on doit pouvoir traiter par exemple `term rest_exp`, d'où l'idée d'utiliser un automate à pile. On construit l'automate à l'aide de la table précédente.

Un automate à pile possède donc une pile, qui peut être ici dans trois états différents :

- prédiction. Si il y a N au sommet de la pile, et qu'on lit le token t en entrée, alors on remplace dans la pile N par ce qu'il y a dans la case (N, t) de la table de transition.
- match (on conserve le token). On mange le token en haut de la pile si il est égal au token de l'input (sinon, erreur).
- fin. Il n'y a plus rien dans la pile. Si la suite des tokens est épuisée, c'est OK. Sinon, erreur.

Comment commence-t-on ? En mettant le start symbol dans la pile.

EXEMPLE 3.2.2 *Essayons de parser avec la grammaire précédente* `(id + id) + id  EOF`.

On représente les différents états de la pile : input $\rightarrow \begin{cases} exp \\ EOF \end{cases} \rightarrow \begin{cases} term \\ rest_exp \\ EOF \end{cases}$

$$\rightarrow \begin{cases} parent_exp \\ rest_exp \\ EOF \end{cases} \rightarrow_{'('} eaten \begin{cases} exp \\ ')' \\ rest_exp \\ EOF \end{cases} \rightarrow \begin{cases} term \\ rest_exp \\ ')' \\ rest_exp \\ EOF \end{cases} \rightarrow_{id} eaten$$

$$\begin{cases} rest_exp \\ ')' \\ rest_exp \\ EOF \end{cases} \rightarrow \ldots EOF \rightarrow THE\ END$$

REMARQUE : Cet algo est déterministe si la grammaire est LL. [1]

3.3 Parsing LR

3.3.1 Analyse de LR(0)

On veut une méthode pour parser la grammaire suivante :

$$\begin{cases} Z \rightarrow E\ \$ \\ E \rightarrow T \mid E + T \\ T \rightarrow i \mid (E) \end{cases}$$

[1] C'est fait pour !

...malgré sa récursivité à gauche. C'est important car dans la vraie vie les grammaires sont écrites comme ça.

Définition 3.3.1 (LR-item)

On considère $N \to \alpha \bullet \beta$: on a déjà reconnu le token α, on espère β. Ici, on ne fait rien tant que le $\bullet$ n'est pas à la fin de la production. On ne réduit qu'à la fin. Si le $\bullet$ est à la fin, on parle de production "reduce", sinon on parle de production "shift".

Remarque : On ne traite pas ici l'étoile car on s'arrange pour mettre les productions sous forme normale de Backus (en utilisant la récursion). On a le même pouvoir d'expression. Du coup, tous les dotted items sont basiques, contrairement à l'analyse lexicale.

On veut parser $i + i\$$. On réalise des transformations sur les productions qui sont similaires à l'analyse lexicale. Ainsi, l'état S_0 est l'ε-fermeture de l'ensemble des productions : $S_0 = \begin{cases} Z \to \bullet\ E\ \$ \\ E \to \bullet\ T \\ E \to \bullet\ E + T \\ T \to \bullet\ i \\ T \to \bullet\ (E) \end{cases}$, et on obtient l'état suivant :

S_0	i		$+$		i		$\$$

Remarque : Pour la fermeture, si on a les règles : $\begin{cases} P \to \alpha \bullet N\beta \\ N \to \gamma \end{cases}$, on ajoute $N \to \bullet\gamma$. Et pour l'initialisation, on a les productions du start symbol avec le $\bullet$ en début de production. Ici $Z \to \bullet E\$$.

Ensuite, on voit le i en entrée, on le mange pour générer l'item $T \to i\bullet$, ce qui est un item réduce (le $\bullet$ est à la fin), on réduit et on se trouve dans la situation suivante :

S_0	T		$+$		i		$\$$

Plus précisément, on crée l'état S_5 : $\left\{ T \to i\bullet \right.$ (on numérote les états pour être cohérent avec l'automate qui va suivre). On peut construire l'arbre syntaxique au vol, quand on remplace i par un T, ce qu'on note :

S_0	$T - i$		$+$		i		$\$$

Le processus est similaire en face du T : l'analyse des états de S_0 conduit à choisir sa deuxième production, ce qui conduit à l'état S_6 : $\left\{ E \to T\bullet \right.$, et donc on a

S_0	$E - T - i$		$+$		i		$\$$

,

Avec nos deux réductions, on a inféré que le i en entrée en fait était un T, qui en fait était en E. Quand on regarde comment consommer ce E, on obtient alors l'état S_1 : $S_1 = \begin{cases} Z \to E \bullet S \\ E \to E \bullet + T \end{cases}$. Pour la première fois il n'y a pas de réduction, c'est un shift item : nous avons trouvé le début d'une production. Attendons d'avoir plus d'info pour déterminer laquelle.

Je vois S_1 devant un $'+'$, ce qui fait que S_3 : $\begin{cases} E \to E\ +\ \bullet\, T \\ T \to\ \bullet\, E \\ T \to\ \bullet\, (E) \end{cases}$ Les deux dernières règles sont obtenues par ε-move, selon la règle d'inférence de la remarque plus haut. On obtient donc ensuite :

S_0	E	S_1	$+$	S_3	i		$\$$

,

On fait un reduce $T \to \bullet i$, et alors

S_0	$E-T-i$	S_1	$+$	S_3	$T-i$		$\$$

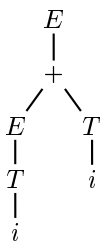
,

A ce stade on a dans la pile $E\ +\ \bullet T$ devant un T, donc on mange le T pour réduire selon $E \to E+T\bullet$:

S_0	A	S_1	$\$$	

,

où A est l'"arbre" suivant :



Notons bien que cet arbre est une instance de non-terminal de type E, il y a un E dans la pile qui pointe sur l'arbre A. Et enfin, on shifte devant le symbole de fin, on réduit selon $Z \to E\$\bullet$, pour obtenir : | S_0 | Z |

Ce qu'on fait, c'est que à partir des règles de la grammaire, on génère l'automate de la figure 3.1 (figure 2.89 du livre de Grune et al).

Les transitions/actions sont résumées dans le tableau suivant :

state \ goto	i	$+$	(	)	$	E	T	action
0	5		7			1	6	shift
1		3			2			shift
2								reduce $Z \to E\$$
3	5		7				4	shift
4								reduce $E \to E+T$
5								reduce $T \to i$
6								reduce $E \to T$
7	5		7			8	6	shift
8		3		9				shift
9								reduce $T \to (E)$

Pour parser, on suit ensuite les règles suivantes :

- **Shift** : prendre le premier token de l'entrée et le mettre en haut de la pile. Regarder dans la table le nouvel état qui correspond à la case (old-state, token), et empiler le résultat.
- **Reduce** avec $N \to \alpha$. On vire de la pile tout ce qui concerne α (les symboles de α et les états qui les suivent immédiatement). On empile N et le nouvel état trouvé dans la table à la case (top-state, N) où top state

est l'état le plus haut dans la pile après l'opération précédente. On a ainsi déterminé le nouvel état.

REMARQUE : Ici bien remarquer que la pile est mixte états, tokens et même non terminaux de la grammaire!

En exercice, refaire sur l'exemple précédent l'algorithme en acceptant lorsque la pile contient l'état initial et le start symbol. Si on n'est pas dans un tel état de pile à la fin de l'input, on refuse l'input :

Pile	Input	Actions
S_0	$i + i\$$	shift
$S_0\ i\ S_5$	$+i\$$	réduire par $T \to i$
$S_0\ T\ S_6$	$+i\$$	réduire par $E \to T$
$S_0\ E\ S_1$	$+i\$$	shift
$S_0\ E\ S_1\ +\ S_3$	$i\$$	shift
$S_0\ E\ S_1\ +\ S_3\ i\ S_5$	$\$$	réduire par $T \to i$
$S_0\ E\ S_1\ +\ S_3\ T\ S_4$	$\$$	réduire par $E \to E + T$
$S_0\ E\ S_1$	$\$$	shift
$S_0\ E\ S_1\ \$\ S_2$		réduire par $Z \to E\$$
$S_0\ Z$		stop-accept

DÉFINITION 3.3.2

LR(0) est une grammaire qui marche pour l'algorithme précédent, *i.e.* l'automate est sans conflit. Par exemple, les expressions bien parenthésées sont donc LR(0).

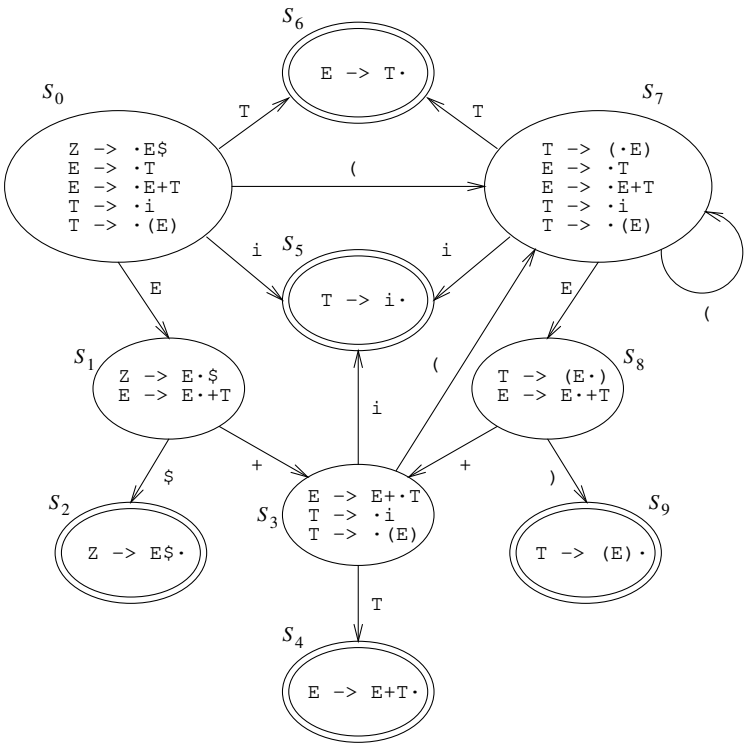


Figure 2.89 Transition diagram for the LR(0) automaton for the grammar of Figure 2.85.

FIG. 3.1 – Automate LR(0).

Problèmes :

- Si on rajoute à la grammaire précédente les tableaux, en ajoutant pas exemple la production suivante : $T \to i \mid i[E]$, on aura un conflit **shift/reduce** lorsque $T \to i\bullet$ et $T \to i \bullet [E]$. On ne sait pas si on doit réduire (on a trouvé un terme) ou shifter (on attend un élément de tableau).
- Si dans la grammaire on ajoute les assignements : $\begin{cases} Z \to V := E \mid E \\ V \to i \end{cases}$, on a un conflit **reduce/reduce** car on a simultanément $V \to i\bullet$ et $T \to i\bullet$ à partir de l'exemple précédent, et on ne sait pas selon quelle règle réduire.

3.3.2 Analyse SLR(1)

Une première solution "simple" est de réduire avec les ensemble FOLLOW. Dès lors, on obtient la définition de SLR(1) ("simple") : c'est une grammaire qui marche avec l'algorithme précédent où l'on ne réduit $N \to \alpha$ que si l'input qui arrive est dans l'ensemble $FOLLOW(N)$. Pour la grammaire considérée, le calcul des FOLLOW des non-terminaux donne :

non-term	follow
E	$\{')', '+', \$\}$
Z	$\emptyset$
T	$\{,')'\ ,\ '+'\ ,\ \$\}$

On réalise donc le tableau suivant ($s3$ signifie que l'on shifte et on va dans l'état 3, $r6$ signifie que l'on réduit avec la 6-ième règle) :

state \ goto	i	+	(	)	\$	E	T
0	s5		S7		s1		s6
1		s3			s2		
2							
3	s5		s7				s4
4		r3		r3			
5		r4		r4			
6		r2		r2			
7	s5		s7			s8	s6
8		s3		s9			
9		r5		r5			

Dans l'exemple de l'extension avec les tableaux, on évitera alors le conflit pour $\begin{cases} T \to i\bullet \\ T \to i \bullet [E] \end{cases}$. En effet, on ne réduit que pour les follows de T (il faut rajouter] à l'ensemble FOLLOW(T) précédent), et en présence d'une parenthèse ouvrante il n'y a plus d'ambiguïté, on shifte.

3.3.3 Analyse LR(1)

On va dans ce paragraphe considérer la grammaire suivante : $\begin{cases} S' \to S\ \$ \\ S \to A \mid xb \\ A \to aAb \mid B \\ B \to x \end{cases}$

Entre parenthèses, elle génère le langage $a^n x b^n$, pour tout $n \geq 0$ union le mot xb. On va montrer qu'elle n'est pas SLR.

Dans la figure 3.2 (figure 2.96 du livre de Grune et al.), les ensemble FOLLOW des productions considérées sont notées entre accolades : {...}. Dans cette même figure, on voit qu'on a un conflit.

Pour cela, le génial Knuth a inventé les follow des items : on va accepter le réduce pour la règle $N \to \alpha\{\sigma\}$, seulement pour un sous ensemble σ de $FOLLOW(N)$. La nouvelle règle d'inférence pour les ε-move est maintenant : INIT consiste à ajouter $\{\$\}$ à la règle $S \to \ldots$; et si $P \to \alpha \bullet N\beta\{\sigma\}$, rajouter pour toute production $N \to \gamma$ l'item $N \to \bullet\gamma\ \{FIRST(\beta\sigma\}$. Par définition, l'ensemble $FIRST(\beta\sigma) = FIRST(\beta)$ si ε n'appartient pas à $FIRST(\beta)$, sinon c'est $FIRST(\beta) \setminus \{\varepsilon\} \cup FIRST(\sigma)$.

La figure 3.3 (figure 2.97 du livre de Grune et al.) réalise cette transformation et on voit bien que l'on a bien reconnu le langage sans conflit. Un tel langage est appelé $LR(1)$.

REMARQUE : Le problème c'est que l'on a bien augmenté le nombre d'états, on invente pour cela les grammaires LALR qui offrent la possibilité de compacter les tables : par exemple ici on peut faire $S_8 = S_{12}$.

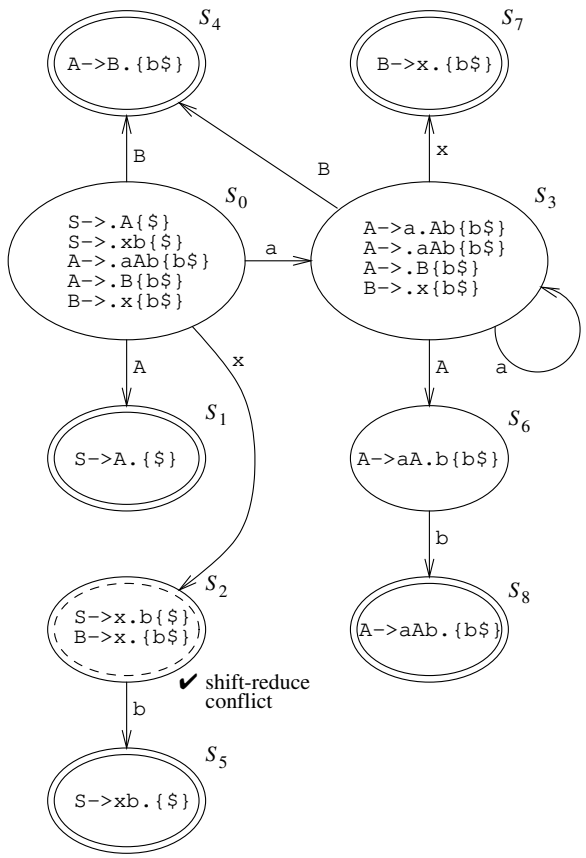


Figure 2.96 The SLR(1) automaton for the grammar of Figure 2.95.

FIG. 3.2 – Automate SLR(1).

132 SECTION 2.2 From tokens to syntax tree − the syntax

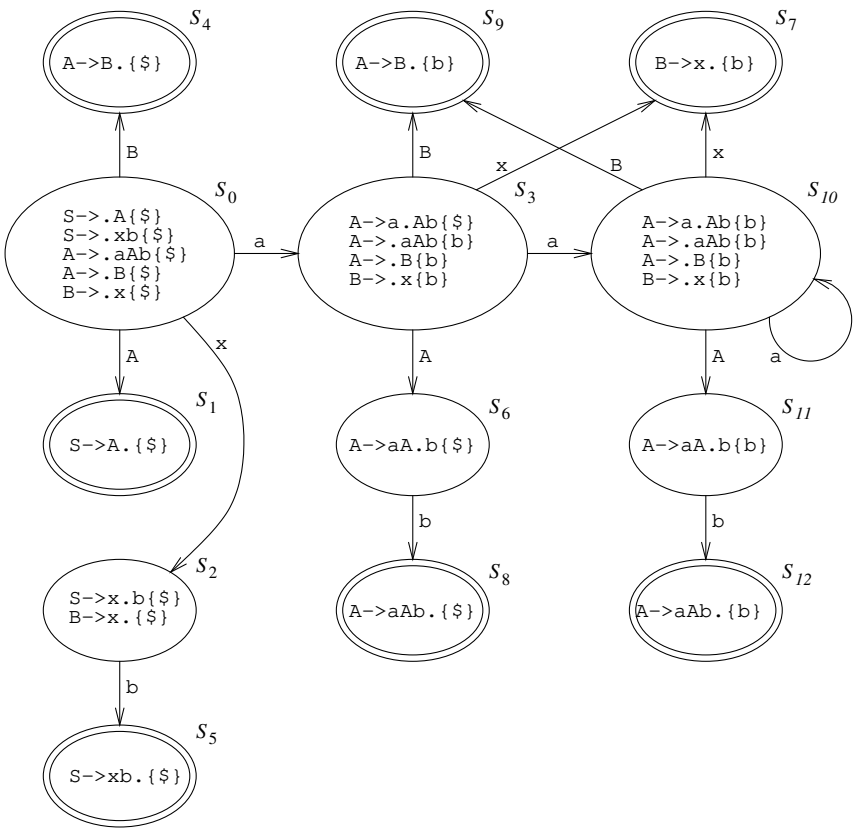


Figure 2.97 The LR(1) automaton for the grammar of Figure 2.95.

FIG. 3.3 – Automate LR(1).

Chapitre 4

Attributs

4.1 Exemples et notions de base

On considère ici la grammaire suivante :

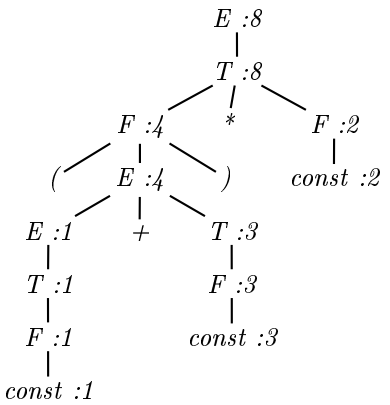
$$G = \begin{cases} E \to E + T & (1) \\ E \to T & (2) \\ T \to T * F & (3) \\ T \to F & (4) \\ F \to -F & (5) \\ F \to (E) & (6) \\ F \to const & (7) \end{cases}$$

On va "décorer" cette grammaire par des assertions, des calculs, du code (pour calculer des valeurs) .... Ici on le fait "à la Yacc", ie on rajoute du code C à droite de la règle afin de faire ce que l'on veut, i.e. ici calculer la valeur des expressions. On obtient :

$$G = \begin{cases} E_1 \to E_2 + T & \{E_1.val \leftarrow E_2.val + T.val\} \\ E \to T & \{E.val \leftarrow T.val\} \\ T_1 \to T_2 * F & \{T_1.val \leftarrow T_2.val * F.val\} \\ T \to F & \{T.val \leftarrow F.val\} \\ F_1 \to -F_2 & \{F_1.val \leftarrow -F_2.val\} \\ F \to (E) & \{F.val \leftarrow E.val\} \\ F \to const & \{F.val \leftarrow const.val\} \end{cases}$$

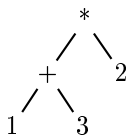
L'attribut VAL est un attribut synthétisé, qui est initialisé par l'analyseur lexical au niveau des feuilles, et qui se calcule en traversant l'arbre de bas en haut.

EXEMPLE 4.1.1 *Lorsque l'on parse l'expression* $(1+3)*2$*, on génère l'arbre :*



Si on veut calculer la valeur, on fait remonter les sous-totaux des feuilles à la racine, en utilisant des attributs.

Profitons de cet exemple pour faire une parenthèse et expliquer brièvement comment on implémente effectivement l'arbre abstrait en machine. Plutôt que de garder tout l'arbre précédent, on se contente de l'arbre suivant :



Pour le construire, on rajoutant du code à droite des règles de la grammaire :

$$\begin{cases} E_1 \to E_2 + T & \{E_1.ptr \leftarrow new\ Opbin(+, E_2.ptr, T.ptr)\} \\ E \to T & \{E.ptr \leftarrow T.ptr\} \\ T_1 \to T_2 * F & \{T_1.ptr \leftarrow new\ Opbin(*, T_2.ptr, F.ptr)\} \\ T \to F & \{T.ptr \leftarrow F.ptr\} \\ F_1 \to -F_2 & \{F_1.ptr \leftarrow new\ Opunaire(-, F_2.ptr)\} \\ F \to (E) & \{F.ptr \leftarrow E.ptr\} \\ F \to const & \{T.ptr \leftarrow new\ Leaf(const.val)\} \end{cases}$$

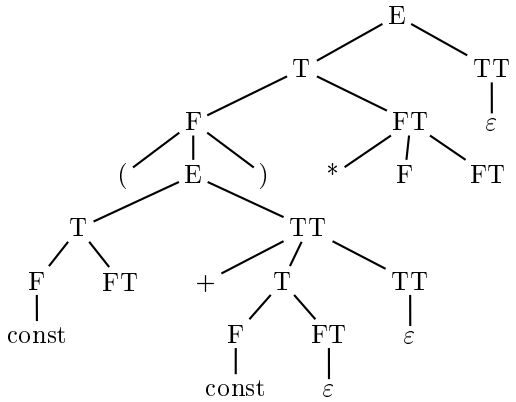
Revenons aux attributs. L'attribut VAL que l'on a utilisé est **synthétisé** car on calcule la valeur du symbole à gauche de la production, en fonction des attributs des symboles à droite de la production. On remonte les informations des feuilles vers la racine. Les attributs synthétisés sont donc parfaits pour une utilisation lors d'un parsing LR.

Les attributs synthétisés ne sont pas suffisants à eux seuls. Certaines caractéristiques de la machine, le contexte, doivent venir d'en haut : on utilise pour cela des attributs **hérités**.

Transformons la grammaire du début (élimination de la récursivité à gauche) pour en faire une grammaire LL :

$$G = \begin{cases} E \to T\ TT & (1) \\ TT \to +T\ TT & (2) \\ TT \to \varepsilon & (3) \\ T \to F\ FT & (4) \\ FT \to *\ F\ FT & (5) \\ FT \to \varepsilon & (6) \\ F \to (E) & (7) \\ F \to const & (8) \end{cases}$$

Sur le même exemple, cela donne le gigantesque arbre [1] :



Evaluer l'expression est devenu plus compliqué. Il faut un ordre d'évaluation compatible avec l'exploration de l'arbre en profondeur et de gauche à droite. On introduit deux attributs, *st* (sous-total, attribut hérité) et *val* (attribut synthétisé comme plus haut) et on écrit : (convention, on numérote de gauche vers la droite lorsqu'il y a ambiguïté sur la variable)

$$\begin{cases} (1) & \{TT.st \leftarrow T.val; E.val \leftarrow TT.val\} \\ (2) & \{TT_2.st \leftarrow TT_1.st + T.val; TT_1.val \leftarrow TT_2.val\} \\ (3) & \{TT.val \leftarrow TT.st\} \\ (4) & \{FT.st \leftarrow F.val; T.val \leftarrow FT.val\} \\ (5) & \{FT_2.st \leftarrow FT_1.st; FT_1.val \leftarrow FT_2.val\} \\ (6) & \{FT.val \leftarrow FT.st\} \\ (7) & \{F.val \leftarrow E.val\} \\ (8) & \{F.val \leftarrow const.val\} \end{cases}$$

Définition 4.1.1

Une grammaire **L-attribuée** est compatible avec l'exploration LL du graphe, i.e. en profondeur et de gauche à droite.

Définition 4.1.2

Une grammaire est bien définie s'il existe un schéma d'évaluation qui respecte les dépendances pour tout arbre de la grammaire

[1] Admirez la prouesse typographique

DÉFINITION 4.1.3
Une grammaire est dite **circulaire** si il existe un cycle de dépendances. Elle peut être bien définie quand même (penser à une évaluation qui converge toujours vers un point fixe en un nombre fini d'itérations)

4.2 Un peu de théorie

On considère ici la règle : $A \to BCD$, A, B, C et D possèdent un certain nombre d'attributs hérédités et synthétisés.

Alors $A.synth, B.inh, C.inh, D.inh = f(A.inh, B.synth, C.synth, D.synth)$. A partir des valeurs *courantes* héritées du père, synthétisées des fils, on peut calculer les autres. On en déduit les flèches de dépendances de l'arbre ci-dessous :

Ceci est une grammaire à attribut générale. La question de son évaluation est traitée dans le Grüne/Bal, auquel nous vous renvoyons.

DÉFINITION 4.2.1
Une grammaire $S-$attribuée ne possède que des attributs synthétisés.
Une grammaire $L-$attribuée possède des productions de la forme $N \to M_1 \dots M_n$ avec :

- $synth(N) = f\big(inh(N), synth(Mi), inh(M_i)\big)$
- $inh(M_j) = f\big(inh(N),\ inh(M_i)$ avec $i < j$, $synth(M_i)$ avec $i < j\big)$:

C'est bien compatible avec le flot LL.

4.3 Liens entre les grammaires S- et L-attribuées

Problème : dans une grammaire L-attribuée, les attributs hérités sont passés des pères aux fils ; dans l'analyse LR le père est créé après ses fils. Est-ce compatible ?

- **Solution 1 :** si la grammaire est LALR (C est dans le follow de A_action), on peut remplacer $A \to B\ \{inh(C) = f(synth(B))\}\ C$ par $A \to B\ A_action\ C$ et $A_action \to \varepsilon\ \{inh(C) = f(...)\}$. C'est la solution utilisée par Yacc. Remarquer qu'en LR, les attributs sont évalués en fin de production, *i.e.* quand on réduit.
- **Solution 2** On retarde l'application de la règle jusqu'à ce que l'on puisse l'évaluer.

On va illustrer cette deuxième solution par l'exemple de la liste de déclaration
`int i,j,k;`

La solution naturelle est d'utiliser un attribut hérité; on propage le type entier aux variables :

$$\begin{cases} Decl_l \to Type_Decl(type)\ Decl_id_seq(type) \\ Decl_idseq(inh\ type) \to Decl_id(type) \mid Decl_seq(type)\ ,\ Decl_id(type) \\ Decl_id(inh\ type) \to id(repr) \quad \text{Attribut : } \{add-to-symbol-table(*repr, type)\} \end{cases}$$

Une autre solution avec seulement des attributs synthétisés consiste à générer une liste de variables en attente, et de les marquer toutes quand on découvre enfin leur type :

$$\begin{cases} Decl_l \rightarrow Type_Decl(type)\ Decl_id_seq(type) \\ \text{Attribut : pour chaque } repr \in repr_list, add-to-table(repr, type) \\ \\ Decl_id_seq(reprlist) \rightarrow Decl_id(repr) \\ \text{Attribut : cree une repr-list initialisée à repr} \\ \\ Decl_id_seq(reprlist) \rightarrow Decl_id_seq(old-repr_list)\ ,\ Decl_id(repr) \\ \text{Attribut : reprlist :=concat(old,repr)} \\ \\ Decl_id(synth\ repr) \rightarrow id(repr) \\ \text{initialisation de l'attribut} \end{cases}$$

Chapitre 5

Contrôle statique

Introduction

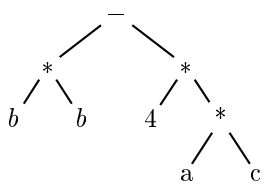
On va partir de l'arbre AST pour faire du contrôle de flot d'exécution, vérifier les types, et aussi la cohérence des déclarations (unicité dans un scope).

5.1 Graphe de flot de contrôle

L'objectif ici est de linéariser l'arbre obtenu, pour obtenir une suite d'instructions élémentaires séquentielles, avec des sauts conditionnels ou inconditionnels. L'exécution du programme se déroulera sur la pile. On va distinguer les cas puis recoller les morceaux ensuite, en laissant les choses difficiles (procédures et objets) pour le chapitre suivant.

5.1.1 Expressions

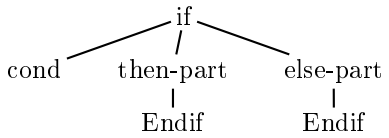
L'expression `b*b-4*a*c` a donné l'arbre :



On fait un parcours en profondeur gauche droite pour faire en sorte de linéariser cet arbre . Le `last_node_pointer` pointe alors sur le signe "-" qui est "vu" en dernier.

5.1.2 If statement

Pour les arbres qui traitent les `if cond then inst1 else inst2`, on va rajouter une case "endif" qui permettra d'avoir un pointeur sur la fin de la séquence, sans savoir si on est rentré dans le "then" ou dans le "else" :



Les "Endif" sont une seule même case. Le `last_node_pointer` pointe alors sur cette case "Endif".

Règles du jeu :
Pour générer du code assembleur, on s'autorise les choses suivantes :
- on veut du code linéaire,
- on utilise le jump simple (GOTO label), le jump indirect (GOTO label_register) et le jump conditionnel (IF cond_reg GOTO label ou IFNOT).

Ce qu'on génère est donc ici :

```
cond_register := expr
IF NOT cond_register GOTO false_label
  --- code for true sequence ----
GOTO end_label
false label:
  --- code for false sequence ---
end label:
...
```

5.1.3 Case statement

On traite ici le cas :

```
CASE case_exp IN
  I1:seq1
  I2:seq2
...
  In:seqn
  DEFAULT:else_statement
END-CASE
```

On distingue les cas suivant le nombre d'alternatives du "case" :
- Si il y a peu d'alternatives possibles, on utilise des GOTO :
```
tp:=case_exp
  if tp=I1 GOTO label_1
  ...
  if tp=In GOTO label_n
  GOTO label_else

  label_1: --code for seq1-- GOTO label_next
  label_2: --code for seq2-- GOTO label_next
  ...
  else: --code for else_seq--
  label_next :
      ...
```
- Sinon, on utilise une table de hachage. C'est toutefois difficile à faire.

5.1.4 Boucles For

On traite ici le cas :

```
FOR i:=lb to ub step step_size statement
```

Attention, il y a plein de pièges :

- On ne peut pas faire un test de sortie sur le plus grand entier de la boucle, il y a un problème si par exemple le dernier entier est MAXINT. Solution : on calcule un nb d'itérations qui décrémente.
- Attention au step qui peut être négatif ou positif
- Attention au step size.

C'est parti :

```
i:=lb

if step_size=0 GOTO error
if step_size<0 GOTO neg_label
if i>ub GOTO end_label

loop_count=(ub-i) DIV step_size + 1
GOTO loop_label

neg_label:
    if i<ub GOTO end_label
    loop_count=(i-ub) DIV (-step_size) +1

loop_label:
    -- code for statement --
    dec(loop_count)
    if loop_count=0 GOTO end_label
    i:=i+step_size
    GOTO loop_label

 end_label:
    ...
```

5.2 Contrôle de types

5.2.1 Scopes

Il y a trois mécanismes de stockage :

- les données statiques sont en mémoire (variables globales). Leur adresse absolue est conservée tout au long de l'exécution du programme,
- les objets sur la pile alloués LIFO (exemple : les procédures),
- les objets sur le tas alloués et désalloués dynamiquement.

- **Identificateurs et espace de nommage** : en C par exemple, il y a 3 espaces dans lesquels de baladent les identificateurs : un pour struct, enum, union, un pour les labels, et un autre pour le reste (variables, routines, identificateurs) ; ce dernier espace est nommé *general name space*. Lorsque l'on déclare :

```
struct one_int{
   int i
}i;
```

alors `i.i` a un sens : le premier "i" est global, il appartient au general name space, et on l'identifiera comme variable de type `struct one_int`. Le deuxième "i" est identifié dans l'espace de nommage propre à la structure.

- **Scope = portée**. Au cours du programme :
  - on empile un nouvel élément vide à l'entrée d'un scope,
  - on lui rajoute les identificateurs déclarés
  - les identificateurs utilisés sont recherchés sur la pile à partir du haut. Si on ne trouve pas dans le scope courant, on cherche dans le scope d'en dessous, *etc.*
  - à la fin du scope, on détruit l'élément et les identificateurs associés dans ce scope.

  Par exemple, en C, le niveau 0 contient les identificateurs des librairies, le level 1 les déclarations des routines, le level 2 les paramètres formels des routines, le level 4 les éventuels sous-blocs , ...Sur l'exemple :

```
void rotate(double angle){
   ...
}

void paint(int left,int right){
   shade matt,signal //signal appartient à la librairire aussi
                     //shade est un nouveau type
   ...
   {counter right,wrong,...}
}
```

  au moment où on exécute paint, le niveau 0 contient signal, printf ; le niveau 1 contient rotate, paint, le niveau 2 contient left, right (mais pas angle), le niveau 3 contient matt et signal, le niveau 4 wrong et right. En fait, on ne stocke pas par niveaux, on stocke pour chaque variable une liste chaînée contenant les niveaux auxquels elle appartient, dans l'ordre décroissant.

5.2.2 Vérification de types

Il faut gérer les trucs suivants :

- **Déclarations de type** : elles peuvent être de deux sortes :
  explicites : `type int-array = ARRAY[1..1515] of integer`
  implicites : `var a : ARRAY[1..1515] of integer` d'où il faut inférer : `type tp_ln_35 = ARRAY[1..1515] of integer` et `var a:tp_ln_35`.

- **Forward declarations** ou déclarations récursives : par exemple

```
TYPE ptr_list = POINTER TO list
TYPE list = RECORD
               Elt:integer
               Next:ptr_list
            END RECORD
```

Solution : on construit au vol une table des types et on teste la cyclicité.

- **Coercitions** C'est par exemple *real* $a := 2$ et *complex* $z := 2$. Il faut faire attention car c'est contexte-dépendant : par exemple on veut que le résultat de `5+3.14` soit un réel mais pas le résultat de `2+2`.

- **Vérification de nature** : lorsque l'on fait une affectation, `dest:=source`, la destination veut une "L-valeur" (*i.e.* une adresse) alors que la source est une "R-valeur" (*i.e.* un contenu). Une variable est suivant les moments une L-valeur ou une R-valeur, mais le plus souvent lors de la déclaration c'est une L-valeur. Lorsque l'on fait $p := q$, on déréférence q pour avoir une adresse, que l'on fournit ensuite à p : on écrit donc :
  ```
  load_mem(q,R_1)
  store_reg(R_1,p)
  ```
  (on fait $R_1 := *(\&q)$ et non pas $R_1 := q$) ! Donc, si on a a gauche de l'affectation une L-valeur et aussi à droite, pas de problème, de même si on a des deux côtés une R-valeur. Par contre si on a à droite une R-valeur et à gauche une L-valeur, on utilise l'opérateur Déref, alors que on pleure dans le dernier cas. Pour gérer tout ça, on se trimballe un attribut synthétisé qui vaut L ou R. On a aussi :
  – const, id (non variable), $\&L$, $R + R$, $L := R$ sont des R-valeurs,
  – id (variable), $*R$ sont des L-valeurs,
  – V[R] et V.chmp sont des R ou des L-valeurs suivant ce que vaut V.

- **Types usuels** Il y a les types :
  – basiques : int, char, float, double
  – énumeration : boolean
  – pointeurs : il faut l'opération de déréférencement : si q est de type T, alors pour $q := *p$ on fait `byte_copie(&q,p,sizeof(T))`.
  – tableaux : `A[i_1,i_2,i_3]` avec chaque i_k vérifiant $LB_k \leq i_k \leq UB_k$. On pose $LEN_k = UB_k - LB_k + 1$. L'endroit en mémoire sera :

  $$baseA + \Big((i_1 - LB_1).LEN_2.LEN_3 + (i_2 - LB_2).LEN_3 + (i_3 - LB_3)\Big).sizeof(el).$$

  – record, routines, objets ...

EXERCICE 5.2.1 *En C : si on a* `int n` *et* `real r`*, que donne* `r = *( (float *) &n )` *?*
On utilise le fait que les pointeurs sur les entiers et sur les réels ont la même représentation en C (une adresse). On suppose que les types `int` *et* `float` *occupent le même nombre d'octets. Alors l'instruction interprète les bits de n comme si c'était un réel (dangereux vu la différence des conventions de représentation !).*

Chapitre 6

Procédures et objets

6.1 Notion d'Activation record

La figure 6.1 représente la pile lors de l'exécution d'une procédure (c'est la figure 6.22 du livre de Grune et al.)

Explications :

- Pour des raisons historiques, la pile grossit vers les basses adresses.
- Le **dynamic link** est un pointeur sur l'appelant (son FP).
- Le **frame pointer** (FP) est le pointeur qui pointe vers la section des variables locales. Il permet d'identifier les objets de la procédure. Son utilisation se fait de la façon suivante : pour accéder au paramètre numéro k, on fait $FP + offset(k)$, pour accéder aux variables, on fat $FP - offset(char)$ par exemple.
- Le **static link** est un pointeur sur (le FP de) la routine lexicographiquement [1] englobante.
- Le **return address** ou adresse de retour de la procédure.

Un exemple d'imbrication des procédures en Pascal : A l'intérieur du code de la procédure A, on insère deux procédures : B puis E. A l'intérieur de B, on code deux procédures C et D qui ne sont accessibles que lorsque l'on est en train d'exécuter le code de B, évidemment. Si on appelle le code B lors de l'exécution du code E, alors on dit que la routine lexico-englobante de B est A, alors que la routine appelante de B est E. Alors on doit aller voir dans A les éventuelles utilisations de variables qui sont utilisées mais pas déclarées dans B, donc on doit pointer sur le FP de A. D'où l'utilisation du static link.

Remarque : On a besoin de l'adresse de retour ET du dynamic link : en effet, si une procédure est récursive, le return address reste le même mais le dynamic link change pour savoir où on en est.

Remarque : D'où vient le nom "lexicographiquement englobante" ?

```
for i1 to n
    for i2 to n
        for i3 to n
```

[1] voir plus loin pour une explication de ce terme

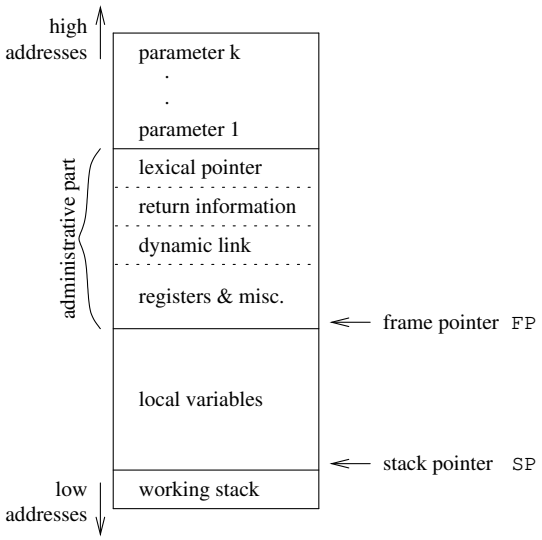


Figure 6.22 Possible structure of an activation record.

FIG. 6.1 – Activation record d'une procédure.

```
        f(A[i1,i2,i3])
    g(B[i1,i2])
    for i3 to n
        h(C[i1,i2,i3])
for i4 to n
    i(E(i1,i4)
```

On touche à `B[2,12]` avant la case `A[3,4,5]` car $(2,12) <_{lex} (3,4)$.

6.2 Appels imbriqués

La figure 6.2 (c'est la figure 6.30 du livre de Grune et al.) décrit ce qui se passe sur les Activation Records lorsque l'on appelle une routine R à l'intérieur de la procédure Q.

Quelques explications :

- AVANT l'appel :
  - Si R est une feuille dans l'arbre de l'exécution, le simple pointeur sur le code de R suffit (routine address).
  - Sinon, on a besoin du lien statique de R (cela sert si on cherche une variable qui n'est pas apparue dans R par exemple, on cherche alors

dans les englobants de R).
  - Le reste se passe de commentaires.
- APRES l'appel :
  - Le return address de R est le bout du code de Q non encore effectué.
  - Le dynamic link de R pointe vers l'Activation Record de Q.
  - Le lexical pointer de R n'a pas bougé.

6.3 Détails d'un appel

6.3.1 Qu'est-ce-qu'on appelle ?

Il faut faire attention à ce qu'on appelle. Par exemple, en Java, si on appelle `toto(int a, int b)` on recherche le premier scope qui matche les deux types des paramètres. On ne cherche donc pas uniquement le nom de la procédure.

6.3.2 Comment on fait ?

La figure 6.3 (c'est la figure 6.45 du livre de Grune et al.) décrit ce qui se passe sur la pile pour une procédure appelante et la procédure appelée :

Quelques explications :
- Le **working area** est utilisé par le compilateur pour mettre des temporaires (calculs, déclarations de types implicites...).
- Le **dynamic part** sert lorsque dans la procédure on crée quelque chose dynamiquement et que l'on libérera juste avant la fin de la procédure. Une telle partie dynamique n'existe pas dans les Sparc. En effet, c'est ennuyeux car on ne connaît pas sa taille à l'avance. A la place, on fera une allocation sur le tas.

Passage par valeur/adresse :
- En Fortran, on passe tout par adresse.
- En Pascal, on passe tout par valeur sauf lorsque l'on utilise le mot clef `var` qui fait passer par adresse (utile lorsqu'on a des tableaux que l'on veut modifier).
- En Java, on passe les références par valeur (*i.e.* une copie de la référence) sauf les types de base. L'utilité de passer une copie de la référence est que l'on peut faire joujou avec sans détruire le pointeur initial.
- En C on passe tout par adresse.

6.4 Quelques remarques sur les objets

La question que l'on se pose ici est celle de l'appel des méthodes d'un objet. Par exemple, dans toute la suite on prend un objet A qui possède les champs a_1 et a_2, ainsi que les méthodes $m_1 - A$ et $m_2 - A$.

Alors si on note a une instance de l'objet A, si on fait l'appel à la méthode $a.m_2 - A(3)$, on peut remplacer cela par (en C) :

```
(def)   m2-A(class A *this,int i) { ... )

(appel) m2-A(&a,3);
```

Dans la suite on va voir comment on gère en compilation quelques unes des caractéristiques des langages objets.

6.4.1 Héritage

Si la classe B hérite de la classe A, cela ne pose pas de problème pour la définition de nouveaux attributs spécifiques à la classe B : champs et nouvelles méthodes : alors B possède par exemple les champs $a1$, $a2$ et $b1$, ainsi que les méthodes $m_1 - A$, $m_2 - A$ et $m3 - B$, les deux premières méthodes étant héritées.

6.4.2 Redéfinition des méthodes

B peut redéfinir la méthode m_2 (qui a pu par exemple seulement être déclarée - par exemple si A est une classe abstraite). Il n'y a toujours pas de problème, on fait :

```
m2-A-A(class A*,...)

m2-A-B(class B*,...)
```

6.4.3 Polymorphisme

Pour pouvoir réaliser la "conversion" suivante :

```
class B *b = ...

class A *a = b
```

On réalise :

```
class A *a = convert_ptr_to_B_to_ptr_to_A(b);
```

Alors, quand on appelle `m2-A(a)`, on appelle quoi ? Eh bien, cela dépend :

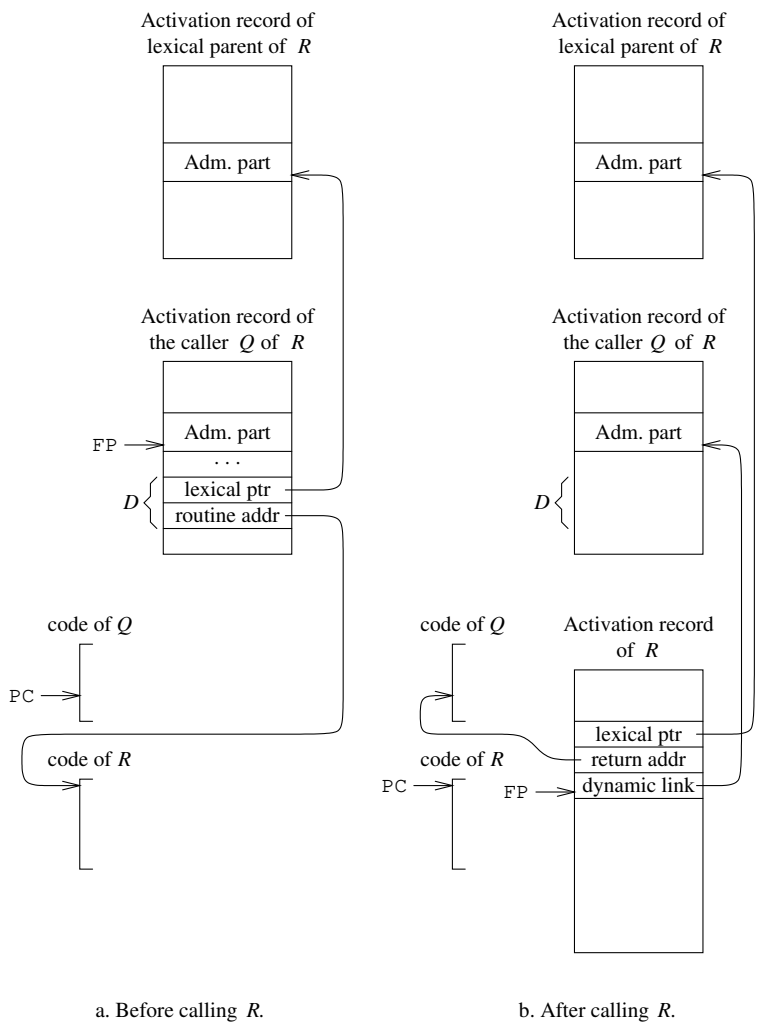
- si le langage est à liaison "dynamique" (dynamic binding), alors on considère a à cet instant comme une instance de B, donc on appelle la fonction `m2-A-B`,
- si c'est "statique", alors on appelle la fonction `m2-A-A`.

Dans le premier cas (généralement choisi par les langages), on doit recoder :

```
void m2-A-B(class A* this, int i) {

   class B* this =  convert_ptr_to_A_to_ptr_to_B(this);

   ---- corps de m2-A-B -----

}
```

6.5 Langages fonctionnels

On verra en TD quels sont les problèmes liés aux langages fonctionnels : portées, définitions de fonctions, unification de types, évaluation paresseuse, ...



a. Before calling R. b. After calling R.

Figure 6.30 Calling a routine defined by the two-pointer routine descriptor D.

FIG. 6.2 – Schéma pour l'appel d'une procédure.

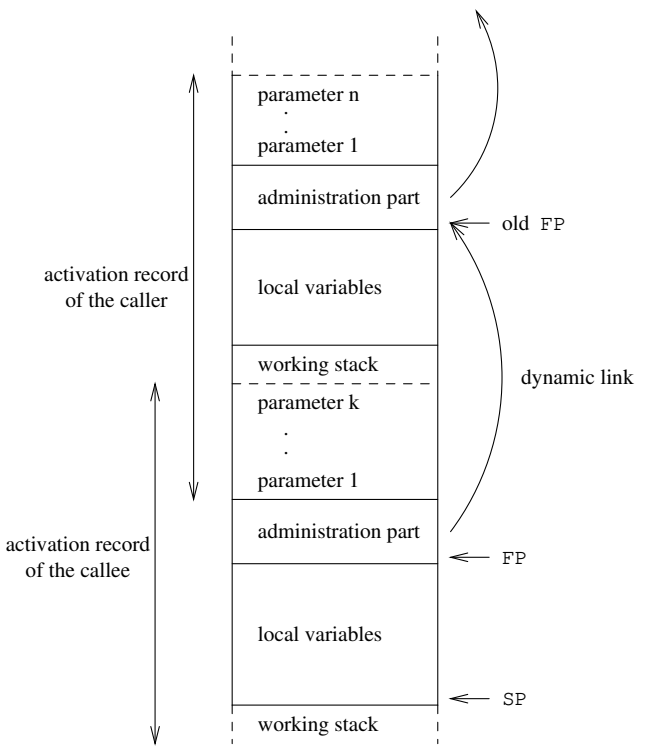


Figure 6.45 Two activation records on a stack.

FIG. 6.3 – Schéma pour l'appel : activation record de l'appelante et de l'appelée.

Chapitre 7

Génération de code

7.1 Introduction

Dans ce chapitre, on génère des bouts de code à partir de l'AST, et on refait le lien avec le chapitre précédent. Il y a plusieurs problèmes à résoudre simultanément :

- Pour quelles parties de l'arbre génère-t-on le code ?
- Avec quelles instructions ? (*i.e.* plusieurs choix possibles selon que les opérandes sont en registre ou en mémoire)
- Gestion des registres
- Ordre des instructions

On va générer une structure de données intermédiaire (graphe) avec :

- des nœuds d'administration (ex : déclarations)
- des nœuds de flots de contrôle (ex : if then else)
- des expressions : objet principal des optimisations.

Dans le chapitre précédent on a étudié comment réaliser les bouts de cette structure de données qui correspondent aux tests, aux while, *etc.*Dans ce chapitre, on va coder les bouts de DAG qui correspondent aux expressions. Un programme sera ensuite un graphe de "basic blocks".

Définition 7.1.1
Un **basic block** est un ensemble maximal de nœuds tel que on commence toujours par le début et on sort toujours par la fin, en fait la première "phrase" (statement) est un label, la dernière un jump et il n'y a pas de goto au milieu.

On va coder les bouts de code correspondant aux expressions pour obtenir des basic blocks que l'on réunira ensuite.

7.2 Génération avec des machines à base de registres

On suppose dans cette section que l'on dispose d'une machine avec un nombre de registres illimité.

Avec une telle machine, on dispose des instructions de la Figure 7.1 (c'est la figure 4.22 du livre de Grune et al.) et de leurs arbres correspondants à la Figure 7.2 (c'est la figure 4.28 du livre de Grune et al.)

234 SECTION 4.2 Code generation

Instruction		Actions
Load_Const	c,R_n	R_n:=c;
Load_Mem	x,R_n	R_n:=x;
Store_Reg	R_n,x	x:=R_n;
Add_Reg	R_m,R_n	R_n:=R_n+R_m;
Subtr_Reg	R_m,R_n	R_n:=R_n−R_m;
Mult_Reg	R_m,R_n	R_n:=R_n*R_m;

Figure 4.22 Register machine instructions.

FIG. 7.1 – Instructions de la register-based machine.

Noter que `Load_Mem x,R` pour l'action `R := x` signifie en fait `Load_Mem &x,R` pour l'action `R := *(&x)`.

Par exemple, si on veut réaliser l'opération `p:=p+5`, on fait :
- Load-Mem p, R_1
- Load-const 5, R_2
- Add-Reg R_2, R_1
- Store-Reg R_1, p

On en déduit un algorithme de génération de code :
- parcours en profondeur du sous-arbre contenant l'expression.
- utiliser une pile de registres.
- une sous-expression stocke son résultat dans un registre-cible à l'aide de registres auxiliaires.

Algo :

```
gen_code(node,Target)
{
   case Constante : Load-const val, R_Target;
        Var       : Load-mem val, R_Target;
        Add       : gen_code(node left  , Target)
                    gen_code(node right , Target+1)
                    Add-Reg R_(Target+1) R_Target
        ...
}
```

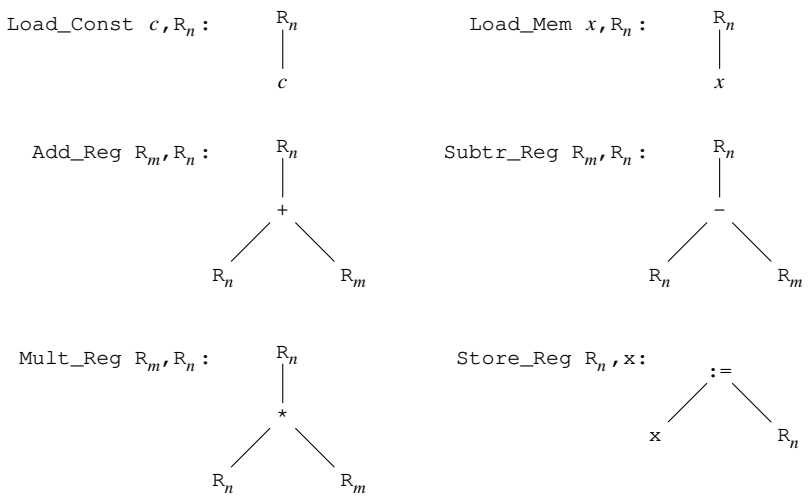
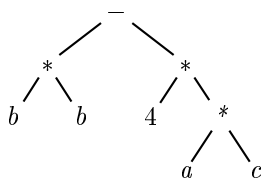


Figure 4.28 The abstract syntax trees for the register machine instructions.

FIG. 7.2 – Arbres des instructions de la register-based machine.

EXEMPLE 7.2.1 *On obtient sur l'exemple de* $b * b - 4 * (a * c)$, *dont l'arbre est*



la séquence :

```
Load_Mem    b,R1
Load_Mem    b,R2
Mult_Reg    R2,R1
Load_Const  4,R2
Load_Mem    a,R3
Load_Mem    c,R4
Mult_Reg    R4,R3
Mult_Reg    R3,R2
Subtr_Reg   R2,R1
```

Ceci dit, sur cet exemple on a été un peu benêt : par exemple, on aurait fait des économies de registre si on avait calculé ac avant 4 : l'idée alors est d'évaluer

le sous-arbre le plus "lourd" en terme de registres. Pour cela, il faut calculer le "poids" des sous-arbres, on fait :

```
  Feuille   -> 1
| Nd(fg,fd) -> if fd=fg then fg+1 else max(fd,fg);
```

C'est l'algorithme de Sethi, qui s'étend aux opérateurs à n arguments. Pour une procédure à n paramètres, si le i-ème paramètre nécessite E_i registres, on trie les E_i par ordre décroissant et on évalue d'abord les paramètres dont le E_i est maximal. On aura besoin de $\text{Max}(E_i + i - 1)$ registres au total.

Le problème c'est qu'en général on n'a pas assez de registres, donc on utilise lorsque l'on n'a pas de registre libre la mémoire. Sur l'exemple, si l'on ne dispose que de 2 registres, on obtient :

```
Load_Mem    a,R1
Load_Mem    c,R2
Mult_Reg    R2,R1
Load_Const  4,R2
Mult_Reg    R2,R1
Store_Reg   R1,T1
Load_Mem    b,R1
Load_Mem    b,R2
Mult_Reg    R2,R1
Load_Mem    T1,R2
Subtr_Reg   R2,R1
```

On appelle cela "spilling", ici T_1 représente une case en mémoire.

7.3 Basic blocks

7.3.1 De l'AST au DDG

DDG = "Data Dependency Graph".

Il y a 3 types de dépendances :

- **dépendance de flot** lorsque on utilise une variable après l'avoir écrite (définie) (il faut que cela se fasse dans cet ordre et pas l'inverse!)
- **anti-dépendance** lorsque on lit une variable avant de l'écrire : `a = x; ... ; x = b` il faut lire x avant sa nouvelle définition pour avoir la bonne valeur en a
- **dépendance de sortie** : deux écritures de suite : si on met toto dans x puis plus tard 4 dans x, il ne faut pas le faire dans un autre ordre.

On va réaliser le DDG correspondant au code suivant :

```
{ int n;
  n := a + 1;
  x := b + n*n + c;
  n := n + 1;
  y := d * n;
}
```

L'AST obtenu pour ce bout de programme est représenté à la Figure 7.3 (c'est la figure 4.42 du livre de Grune et al.), son DDG à la Figure 7.4 (c'est la figure 4.43 du livre de Grune et al.). IMPORTANT : les flèches sont orientées

à l'opposé des dépendances dans le livre de Grune et al., il faut les retourner toutes. Les flèches qui viennent d'en haut sont les variables qui devront être visibles de l'extérieur. Ce n'est pas le cas de n car c'est une variable locale.). Le DDG "nettoyé" est donné Figure 7.5.

254 SECTION 4.2 Code generation

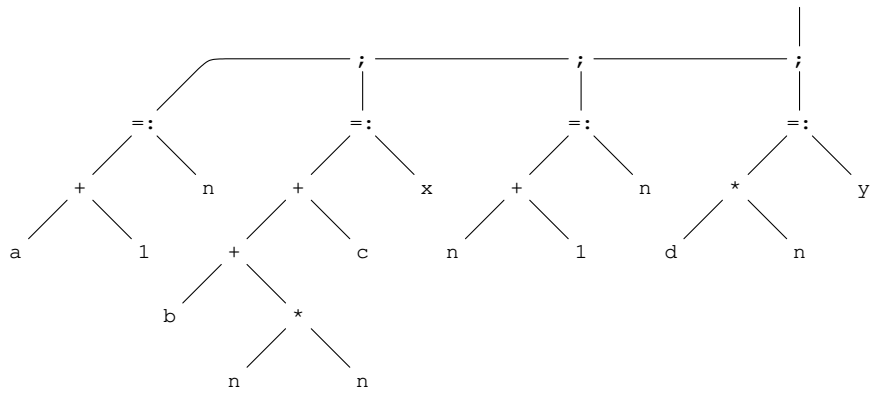


Figure 4.42 AST of the sample basic block.

FIG. 7.3 – AST de l'exemple.

On génère alors une allocation virtuelle des registres ou de la mémoire :

```
pos  triple
 1   a+1
 2   @1 * @1
 3   b + @2
 4   @3 + c
 5   @4 -> x
 6   @1 + 1
 7   d * @6
 8   @7 + y
```

Il ne reste plus qu'à allouer selon les vraies ressources physiques, c'est l'objet du paragraphe suivant.

7.3.2 Du DDG au code : les échelles

On a des vraies machines qui disposent d'opérations registre-registre et mémoire-registre. Avec une telle machine, tout nœud peut (enfin devrait) être écrit avec une seule instruction. Mais il y a plusieurs choix possibles. On ajoute donc des instructions du type `Add_Mem x,R`, `Mult_Mem x,R` aux instructions précédentes.

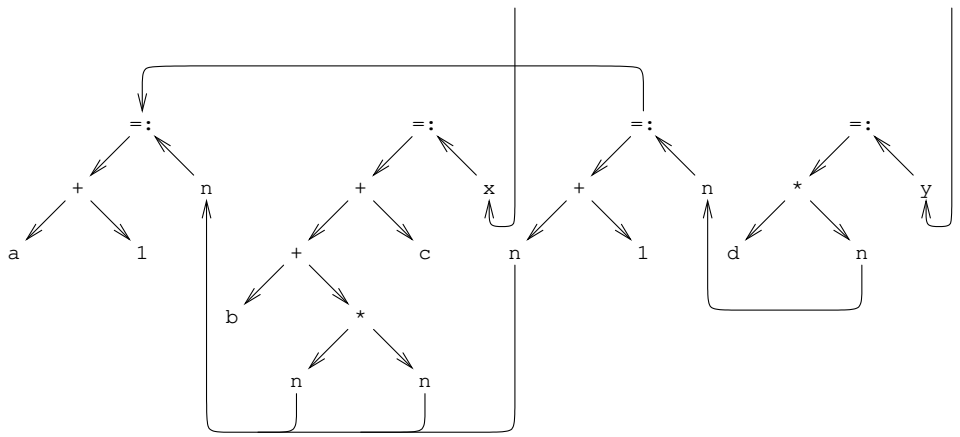


Figure 4.43 Data dependency graph for the sample basic block.

FIG. 7.4 – DDG de l'exemple.

Une "échelle pour un DDG" commence du nœud racine et se poursuit le plus longtemps possible en passant par le fils gauche (sauf si l'opération est commutative auquel cas on peut poursuivre par le nœud droit).

Par exemple, pour le sous-arbre de la Figure 7.6 (c'est la figure 4.49 du livre de Grune et al.), $x, +, +, b$ est une échelle et on génère alors le code donné.

Le résultat de l'évaluation pour chaque échelle est stockée dans un registre spécial (le même pour chaque échelle). Deux questions se posent alors :

- Comment partitionner le DDG en échelles ?
- Comment générer le code des échelles ?

Algo :

- Utiliser des pseudo-registres
- Garder un registre spécial R_1 pour le résultat de chaque échelle.
- Tant qu'il reste des nœuds à traiter :

  1. Trouver une échelle S dont tous les nœuds ont au-plus une dépendance en sortie.

  2. Si une opérande d'un nœud N de S n'est pas une feuille mais un nœud M alors :
     - Marquer M comme nouvelle racine.
     - Associer un pseudo-registre R à M (si ce n'est déjà fait).
     - Utiliser R pour le code généré de N.

  3. Générer le code pour S avec R_1 et supprimer S.

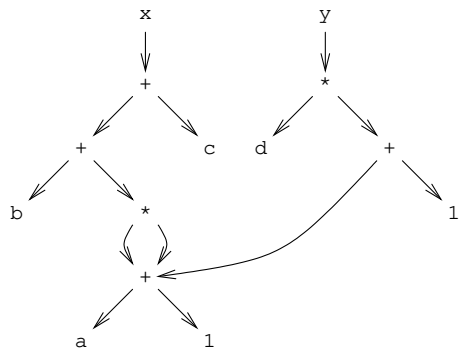


Figure 4.44 Cleaned-up data dependency graph for the sample basic block.

FIG. 7.5 – DDG nettoyé pour l'exemple.

On va réaliser cet algorithme plusieurs fois afin de générer le code pour le DDG de la Figure 7.5. Les échelles utilisées seront au début a- `y * +` et b- `x + + *`

1. : Première étape : pour l'échelle $a-$, l'opérande gauche de + n'est pas une feuille, c'est le nœud + de $a + 1$. Je lui associe le pseudo-registre X_1 et je marque le nœud racine. Donc pour cette échelle, on génère le code A suivant :

```
Load-Reg X1,R1
Add-Cte   1,R1
Mult-Mem  d,R1
Store-Reg R1,y
```

2. : 2ème étape : l'échelle $b-$ utilisée ici est maximale car on ne lui rajoute pas le dernier + qui est source de 3 dépendances dans le DDG (on veut au plus une dépendance). On génère le code B suivant : (bien remarquer que l'on se sert de R_1 aussi ici)

```
Load-Reg  X1,R1
Mult-Reg  X1,R1
Add-Mem   b,R1
Add-Mem   c,R1
Store-Reg R1,x
```

3. : 3ème étape : Il ne reste que le bout d'arbre suivant : On génère alors le code C suivant :

```
Load-Mem  a,R1
Add-cte   1,R1
Load-Reg R1,X1
```

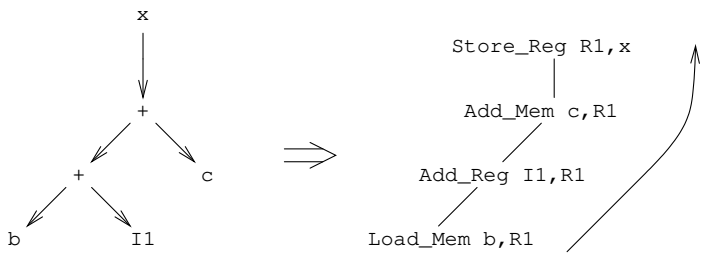


Figure 4.49 Rewriting and ordering a ladder sequence.

FIG. 7.6 – Un exemple d'échelle.

Le code final généré est CBA, dans cet ordre. Ensuite, on renomme $X1$ en le registre R_2. Le problème, c'est que l'on fait des choses redondantes, des "move" (la dernière instruction de C met R1 dans X1, et la première de B met X1 dans R1). Durant la phase d'allocation des registres, on fait en sorte d'éliminer de telles redondances (l'idée est d'évaluer la durée de vie de chaque variable pour mettre deux variables compatibles -leurs durées de vie ne s'intersectent pas- dans le même registre). On obtient alors le code suivant :

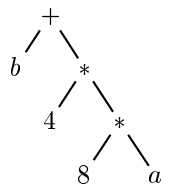
```
Load-Mem  a,R1
Add-cte   1,R1
Load-Reg  R1,R2

Mult-Reg  R1,R2
Add-Mem   b,R2
Add-Mem   c,R2
Store-Reg R2,x

Add-Cte   1,R1
Mult-Mem  d,R1
Store-Reg R1,y
```

Pour l'évaluation des durées de vie, on pourra se reporter au chapitre suivant.

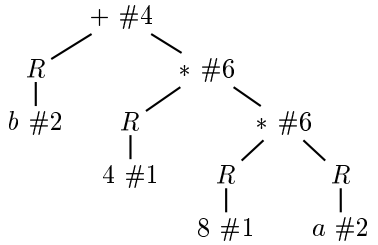
7.4 Sélection des instructions

Il est fréquent que les opérandes puissent être prises en mémoire et en registre. Sur le modèle de la Figure 7.7 (c'est la figure 4.60 du livre de Grune et al.) les instructions ont des coûts différents. Comment alors combiner les différentes instructions pour avoir un coût minimal ?

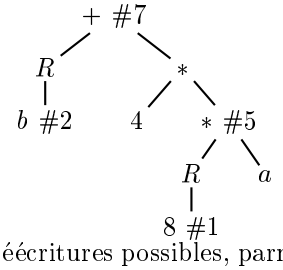
Sur l'exemple suivant :



- La solution "pure registre", qui consiste à prendre les instructions selon la réécriture qui suit donne un coût de 17 :



- Solution 2 : "glouton" partir du haut avec la plus grosse instruction : (coût 14)



- **Pb :** trouver toutes les réécritures possibles, parmi celles-ci, trouver celles de poids minimum.

On va détailler deux solutions, puis une solution optimale qui combine les deux approches.

7.4.1 Bottom-up pattern-matching

On traverse l'arbre de bas en haut, on annote chaque nœud de tous les choix possibles avec le numéro de l'instruction et la place du résultat (registre ou mémoire) selon la syntaxe $\#L \rightarrow location$. On obtient alors l'arbre de la Figure 7.8 (c'est la figure 4.69 du livre de Grune et al.)

REMARQUE : On n'utilise pas ici le fait que + est commutatif. La façon la plus simple de prendre en compte la commutativité d'un opérateur est de rajouter l'instruction symétrique dans le jeu des instructions. Ensuite, on peut générer la suite d'instruction en piochant au hasard dans les instructions autorisées à la racine, puis dans les fils ce qui est encore possible au hasard aussi, *etc.*

7.4.2 Instruction selection by dynamic programming

Parmi toutes les transitions possibles, je cherche les meilleures : l'idée de programmation dynamique est que lorsque dans un nœud on a deux types d'instructions dont l'une est meilleure en terme de coût que l'autre, il n'y a pas de raison pour que plus haut on ait besoin de cette instruction plus chère. On étend

la natation pour conserver les coûts, avec la syntaxe $\#L \rightarrow location@cost$. On obtient toujours sur le même exemple l'arbre de la Figure 7.9 (c'est la figure 4.71 du livre de Grune et al.)

On obtient alors un coût de 13, qui passe à 12 si on tient compte de la commutativité du +.

7.4.3 BURS : Bottom-up Rewriting system

On combine ici les deux solutions précédentes. On va construire un automate à partir des états de base S_1 qui est un état des constantes, qui comprend les deux items $\rightarrow cst@0$ et $\#1 \rightarrow reg@1$, et S_2 qui est l'état des variables, qui comprend les deux items $\rightarrow meme@0$ et $\#2 \rightarrow reg@3$ Alors on construit :

- S_3 :

  $$\begin{array}{ccc} & + & \\ / & & \backslash \\ S_1 & & S_1 \end{array}$$

  Les candidats sont les instructions #3, #4, #7 : on élimine #3 car il y a besoin de mémoire et #7 car il faut que l'une des opérandes soit #7.1. L'état S_3 est donc : $\#4 \rightarrow reg@1 + 1 + 1 = 3$

- S_4 :

  $$\begin{array}{ccc} & + & \\ / & & \backslash \\ S_1 & & S_2 \end{array}$$

  On élimine #7 pour incompatibilité, on a $\#3 \rightarrow reg@1 + 0 + 3 = 4$ et $\#4 \rightarrow reg@1 + 3 + 1 = 5$ que l'on élimine car il coûte trop cher.

- $S_5 = S_1 + S_4$, on obtient $\#4 \rightarrow reg@6$, pour $S_6 = S_1 + S_5$ on obtient $\#4 \rightarrow reg@8$, etc. On obtient donc une infinité d'états, ce qui est gênant.

On remédie au problème précédent en normalisant les coûts : on retranche le coût minimum d'un item de l'état à chaque item de l'état. Alors les états S_3, S_4, S_5, S_6 ont le même item $\#4 \rightarrow reg@0$, ce qui est bien. On obtient alors les états renumérotés de la Figure 7.10 (c'est la figure 4.74 du livre de Grune et al.). Puis on obtient la table de transitions d'un état à un autre, Figure 7.11 (c'est la figure 4.76 du livre de Grune et al.), dans laquelle le - signifie que toutes les colonnes sont identiques pour l'addition, et toutes sauf la première sont identiques pour la multiplication.

Enfin, on génère dans le même temps la table de la Figure 7.12 (c'est la figure 4.77 du livre de Grune et al.) qui décrit les possibilités : constante, mémoire, registre, et avec quelle instruction.

Comment reconstruit-on le code ? On fait d'abord une remontée de l'arbre qui permet de retrouver les états, puis on redescend en générant les instructions. Toujours dans le même exemple, on trouve :

$$\begin{array}{ccccccc} & & & +, S_{03}, \#4 & & & \\ & & / & & \backslash & & \\ & b, S_{02}, \#2 & & & *, S_{12}, \#8 & & \\ & & & / & & \backslash & \\ & & 4, S_{01}, \#1 & & & *, S_{06}, \#8.1 & \\ & & & & / & & \backslash \\ & & & 8, S_{01}, \#1 & & & a, S_{02}, \#2 \end{array}$$

REMARQUE : Il y a des limitations : ce n'est pas vrai que les instructions ont toujours un coût fixe. De plus on ne gère pas le pipeline. Sur les processeurs modernes, il faut ensuite réordonner les instructions en gérant les délais. Quelquefois, on prend en compte le pipeline en mettant des coûts optimistes (comme si on pouvait toujours pipeliner). Si au moment de faire l'instruction on ne peut pas, on introduit alors des NO-OPs.

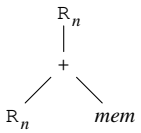
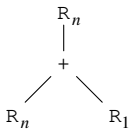
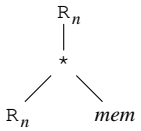
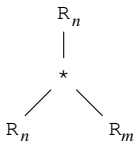
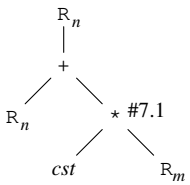
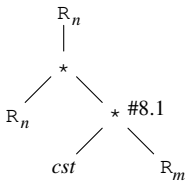
#1		`Load_Const` *cst*, R_n	load constant	cost = 1
#2		`Load_Mem` *mem*, R_n	load from memory	cost = 3
#3		`Add_Mem` *mem*, R_n	add from memory	cost = 3
#4		`Add_Reg` R_1, R_n	add registers	cost = 1
#5		`Mult_Mem` *mem*, R_n	multiply from memory	cost = 6
#6		`Mult_Reg` R_m, R_n	multiply registers	cost = 4
#7		`Add_Scaled_Reg` *cst*, R_m, R_n	add scaled register	cost = 4
#8		`Mult_Scaled_Reg` *cst*, R_m, R_n	multiply scaled register	cost = 5

Figure 4.60 Sample instruction patterns for BURS code generation.

FIG. 7.7 – Jeu d'instruction pour BURS.

SUBSECTION 4.2.6 *BURS code generation and dynamic programming* 281

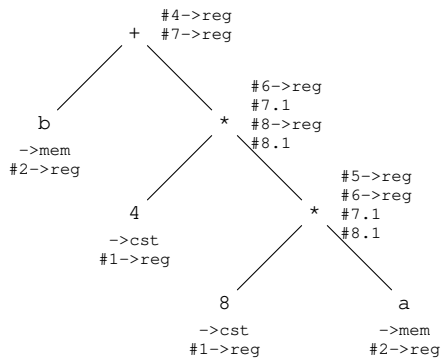


Figure 4.69 Label sets resulting from bottom-up pattern matching.

FIG. 7.8 – Bottom-up pattern matching.

SUBSECTION 4.2.6 *BURS code generation and dynamic programming* 283

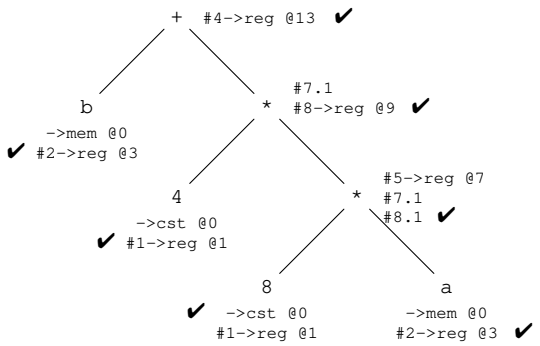


Figure 4.71 Bottom-up pattern matching with costs.

FIG. 7.9 – Coût par programmation dynamique.

```
S00 = { }
S01 = {→cst@0, #1→reg@1}
S02 = {→mem@0, #2→reg@3}
S03 = {#4→reg@0}
S04 = {#6→reg@5, #7.1@0, #8.1@0}
S05 = {#3→reg@0}
S06 = {#5→reg@4, #7.1@0, #8.1@0}
S07 = {#6→reg@0}
S08 = {#5→reg@0}
S09 = {#7→reg@0}
S10 = {#8→reg@1, #7.1@0, #8.1@0}
S11 = {#8→reg@0}
S12 = {#8→reg@2, #7.1@0, #8.1@0}
S13 = {#8→reg@4, #7.1@0, #8.1@0}
```

Figure 4.74 States of the BURS automaton for Figure 4.60.

FIG. 7.10 – Ensemble des états pour BURS.

'+'	S_{01}	S_{02}	S_{03}	S_{04}	S_{05}	S_{06}	S_{07}	S_{08}	S_{09}	S_{10}	S_{11}	S_{12}	S_{13}
S_{01} – S_{13}	S_{03}	S_{05}	S_{03}	S_{09}	S_{03}	S_{09}	S_{03}	S_{03}	S_{03}	S_{03}	S_{03}	S_{03}	S_{09}

'*'	S_{01}	S_{02}	S_{03}	S_{04}	S_{05}	S_{06}	S_{07}	S_{08}	S_{09}	S_{10}	S_{11}	S_{12}	S_{13}
S_{01}	S_{04}	S_{06}	S_{04}	S_{10}	S_{04}	S_{12}	S_{04}	S_{04}	S_{04}	S_{04}	S_{04}	S_{13}	S_{12}
S_{02} – S_{13}	S_{07}	S_{08}	S_{07}	S_{11}	S_{07}	S_{11}	S_{07}	S_{07}	S_{07}	S_{07}	S_{07}	S_{11}	S_{11}

Figure 4.76 The transition table `Cost conscious next state[]`.

FIG. 7.11 – Tables de transition pour BURS.

	S_{01}	S_{02}	S_{03}	S_{04}	S_{05}	S_{06}	S_{07}	S_{08}	S_{09}	S_{10}	S_{11}	S_{12}	S_{13}
cst		--	--	--	--	--	--	--	--	--	--	--	--
mem	--		--	--	--	--	--	--	--	--	--	--	--
reg	#1	#2	#4	#6	#3	#5	#6	#5	#7	#8	#8	#8	#8

Figure 4.77 The code generation table.

FIG. 7.12 – Tables de génération de code pour BURS.

Chapitre 8

Registres

8.1 Liveness Analysis (vivacité)

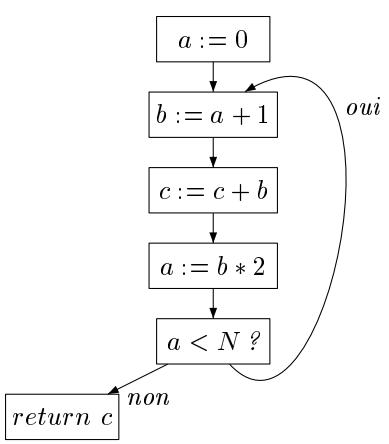
8.1.1 Exemple

Une définition informelle du caractère "vivant" d'une variable pourrait-être la suivante : une variable est vivante si sa valeur courante est utilisée dans la suite. La propriété de "liveness" est donc une propriété qui se calcule "bottom-up", en remontant les arêtes du CFG (Control Flow Graph).

EXEMPLE 8.1.1 *Dans le code suivant :*

```
     a:=0
L1:  b:=a+1
     c:=c+b
     a:=b*2
     if a<N goto L1
     return c
```

On obtient le graphe d'exécution :



Et donc :

- *a : live-out en S_1, live-in en S_2, live-out en S_4 et S_5 et live-in en S_5. a n'est pas live-in en S_3 car bien que parfaitement définie sa valeur courante n'est jamais utilisée (pas en S_3, et redéfinie en S_4).*
- *b : live-in en S_3 et S_4, live-out en S_2, ...*
- *c : live-in partout (donc c n'est pas une variable locale au bloc ou alors elle n'a pas été initialisée).*

Comme les périodes de vie de a et b ne se superposent pas, on peut mettre a et b dans le même registre.

C'est ce processus qu'on va essayer d'automatiser à la compilation.

8.1.2 Equation data-flow

Tour d'abord, quelques définitions :

DÉFINITION 8.1.1 (DEF ET USE D'UN NOEUD)

- def : une expression définit une variable
- use : une occurrence en membre droit l'utilise

EXEMPLE 8.1.2 *Dans l'exemple $def(3) = \{c\}$ et $use(3) = \{c, b\}$.*

A partir de ces définitions, on va obtenir une définition plus formelle des durées de vie des variables :

DÉFINITION 8.1.2

Une variable a est dite **live** sur une arête si il existe un chemin de cette arête vers un usage de a qui ne passe par aucune définition de a.
Une variable est **live-in** sur un nœud si elle est live sur une arête entrante de ce nœud. Une variable est dite **live-out** sur un nœud si elle est live sur une arête sortante

DÉFINITION 8.1.3 (ENSEMBLES OUT ET IN)

L'ensemble *in* d'un nœud est l'ensemble des variables qui sont live-in sur ce nœud. L'ensemble *out* d'un nœud est l'ensemble des variables qui sont live-out sur ce nœud.

On obtient facilement la :

PROPOSITION 8.1.1

$$in(n) = use(n) \cup \big(out(n) \setminus def(n)\big)$$

$$out(n) = \bigcup_{s \in succ(n)} in(s)$$

Pour calculer ces ensembles, on va utiliser un algorithme itératif qui va faire grossir les ensembles jusqu'à converger vers un point fixe. On initialise tous les ensembles $in(n)$ et $out(n)$ à vide au début. Sur l'exemple, on obtient :

state	use	def	in	out	in	out	in	out	in	out	in	out	in	out	in	out
1		a				a		a		ac	c	ac	c	ac	c	ac
2	a	b	a		a	bc	ac	bc	ac	bc	ac	bc	ac	bc	ac	bc
3	bc	c	bc		bc	b	bc	b	bc	b	bc	b	bc	bc	bc	bc
4	b	a	b		b	a	ba	bac	bc	ac	bc	ac	bc	ac	bc	ac
5	a		a	a	a	ac	ac	ac	ac	ac	ac	ac	ac	ac	ac	ac
6	c		c		c		c		c		c		c		c	

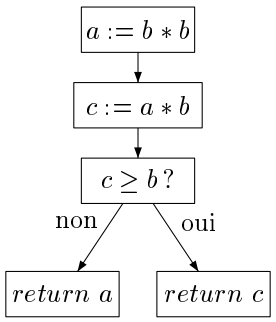
C'est long! Mais en prenant des nœuds dans l'ordre inverse, et en calculant les ensembles out avant les ensembles in, on obtient.

state	use	def	1-out	1-in	2-out	2-in	3-out	3-in
6	c			c		c		c
5	a		c	ac	ac	ac	ac	ac
4	b	a	ac	bc	ac	bc	ac	bc
3	bc	c	bc	bc	bc	bc	bc	bc
2	a	b	bc	ac	ba	ac	bc	ac
1		a	ac	c	ac	c	ac	c

Ceci dit, le théorème de Tarski garantit que la solution trouvée est la même. Enfin, remarquons que l'on cherche le plus petit point fixe. La solution trouvée reste solution si on ajoute d à chaque colonne, mais cela n'a aucun intérêt.

8.1.3 Static vs dynamic liveness

On distingue vivacité statique et vivacité dynamique :



Avec un peu d'analyse, on se rend compte que la flèche "non" n'est jamais utilisée. Statiquement, on dirait de la variable a qu'elle est vivante car on peut s'en servir, mais dynamiquement, c'est faux de dire que a est vivante car on ne s'en servira jamais.

On dit que a est vivante dynamiquement en un nœud N si il existe une exécution du programme qui va de N a une utilisation de a sans passer par une définition. On dit que a est vivante statiquement au nœud N si il existe un chemin allant de N à une utilisation éventuelle de a dans le CFG.

Faire une analyse dynamique est hors de porté, et on se contente d'une analyse statique (plus conservative donc plus exigeante en ressources de type registres) à la compilation.

8.1.4 Graphe d'interférence

Les sommets de ce graphe sont les variables du code. Si a et b sont live en un même nœud du CFG, alors on les relie par une arête. Cette arête symbolise l'impossibilité de mettre les deux variables dans le même registre.

8.1.5 Traitement des move

Les move sont générés lors de l'analyse des blocs. On génère des pseudo-registres distincts qu'on devrait plonger dans le même registre physique, pour éviter des copies inutiles :

```
t:=s; (copy)
    :           <- t et s sont live-out
    :
    x:=s;
    y:=t;
```

Plus précisément,

- Lorsqu'il y a un **non-move statement** (pas une copie) qui définit a et que dans les variables live-out on a $b_1, b_2, \ldots b_j$, alors ajouter les flèches $a \leftarrow b_i$ au graphe d'interférence
- Lorsqu'il y a un **move-statement** : $a \leftarrow c$ qui est une copie, on met toutes les flèches précédentes sauf $a \leftarrow c$ si c est l'un des b_j.

L'idée est que tant que a et c sont la même chose, on ne met pas de flèche dans le graphe.

8.2 Allocation de registres

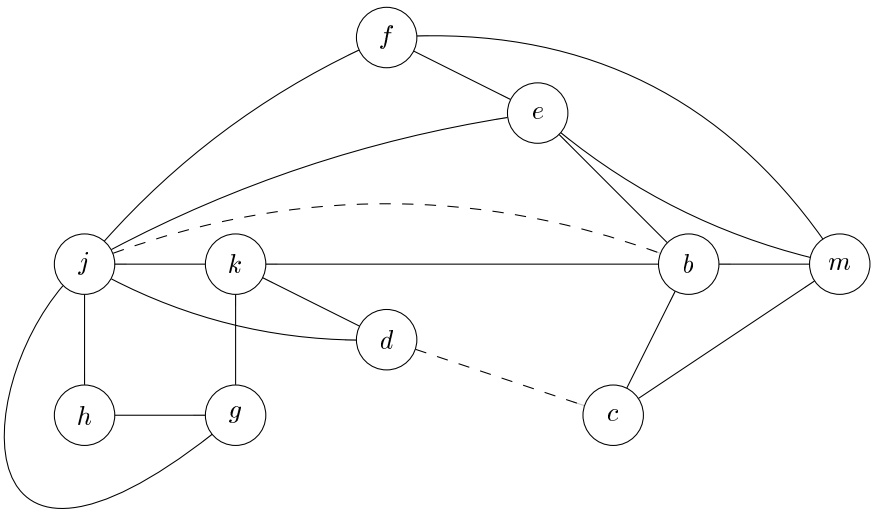
A l'aide de ce graphe, nous allons définitivement allouer les registres, et ce par plusieurs méthodes. Dans toute la suite de cette section, on utilisera le gros exemple suivant :

```
live_in k,j
   g:=*(j+12);
   h:=k-1;
   f:=g*h;
   e:=*(j+8);
   m:=*(j+16);
   b:=*f;
   c:=e+8;
   d:=c;
   k:=m+4;
   j:=b;
live_out d,k,j
```

On va tracer le graphe d'inférence, en remarquant que :

- les variables d,k,j sont live-out en sortie, donc il y a un triangle entre eux.
- j interfère avec g, h, f et e, mais pas avec m qui est son dernier usage avant sa redéfinition.
- il y a deux instructions move qu'on note avec des flèches pointillées.

Et voila enfin le graphe :



Dans la suite, on va essayer de colorier le graphe avec un nombre de couleurs égal au nombre de registres disponibles. Le problème est différent du problème classique qui consiste à utiliser le moins de couleurs possibles. Ici on s'attache à colorier le plus de nœuds possibles avec le nombre de couleurs données. Ceci dit, le problème reste NP-complet. Les nœuds non coloriés seront les variables qui devront aller en mémoire (spill).

8.2.1 Coloriage par simplification

Idée : on commence par éliminer les sommets qui ont moins de K voisins, où K est le nombre de registres. Le procédé est le suivant :

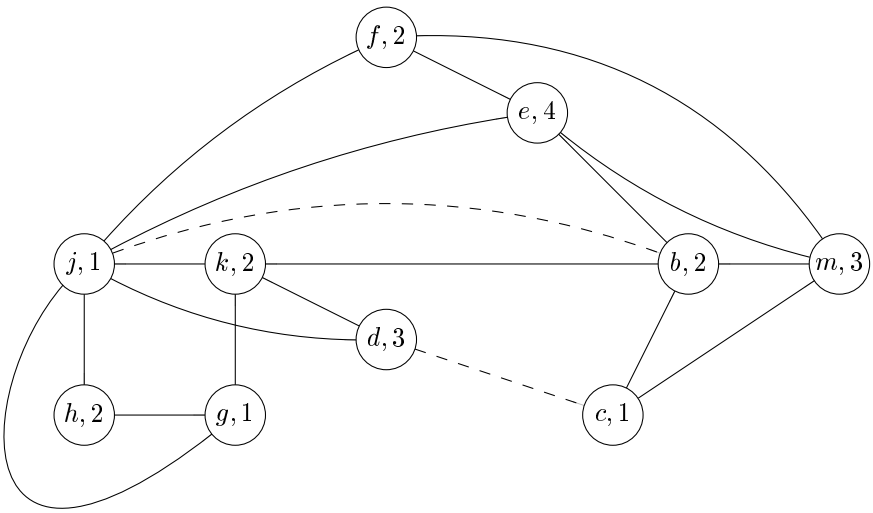
- **Build** Construction du graphe d'inférence $G = (V, E)$.
- **Simplify** Si on a K registres, si on trouve un $v \in V$ non significatif, c'est-à-dire que $deg(v) < K$, on colore le reste et on trouvera forcément une couleur pour v à la fin. (donc on empile ce nœud pour plus tard)
- **Spill** Si il ne reste que des nœuds significatifs, en spiller un, *i.e.* l'empiler en le marquant "Potential Spill". On espère qu'on pourra le colorier quand même parce que plusieurs de ses voisins auront reçu la même couleur.
- **Dépiler et colorier** : si le nœud est non marqué, on est sur que l'on va pouvoir le colorier, si il est marqué "Potential Spill", on essaie quand même.

REMARQUE : On ne s'occupe pas des move ici.

On va reprendre l'exemple du graphe précédent que l'on va colorier avec $K = 4$ couleurs (on dispose de 4 registres) :

- On empile successivement $f, e, m, b, c, d, k, g, h, j$ sans en marquer aucun.
- On doit donc pouvoir tout colorier.
- Pour colorier, l'heuristique utilisée ici est de mettre le plus petit numéro de couleur disponible.

On obtient donc le coloriage : (le chiffre à côté de la variable désigne son coloriage)



Si on essaie de colorier ce graphe avec 3 couleurs, on obtient tout d'abord la pile $c, h, g, k, j(PS), f, d, b, m, e$ et lorsque l'on dépile finalement on peut colorier j qui n'est pas un "actual spill".

REMARQUE : Ici on n'a pas tenu compte des flèches pointillées, *i.e.* des move. Seulement, ces move sont relativement nombreux, alors on va essayer de fusionner certains de ces nœuds move. On va voir dans la suite que c'est assez expérimental.

8.2.2 Traitement des move

La stratégie utilisée est *conservative* : on va fusionner [1] uniquement les nœuds move qui n'handicapent pas la coloration. Pour cela, on distingue dans la littérature deux heuristiques :

- A : On fusionne a et b seulement si le nœud obtenu est de degré inférieur strict à K nombre de registres.
- B : On fusionne a et b si tout voisin de a soit est non significatif, soit interfère déjà avec b.

En général, on alterne *coalesce* et *simplify* de manière purement empirique :

a) Build : en marquant les nœuds move.

b) Simplify : on simplifie les non move de degré $< K$.

c) Coalesce

d) Freeze : on abandonne la fusion d'un move et on supprime l'arête pointillée correspondante.

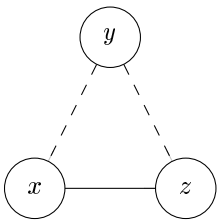
e) Spill si échec.

f) Select.

En pratique, on attend que l'on ne puisse plus faire de fusion pour passer à l'étape d. Mais on n'a pas de mesures autres qu'expérimentales.

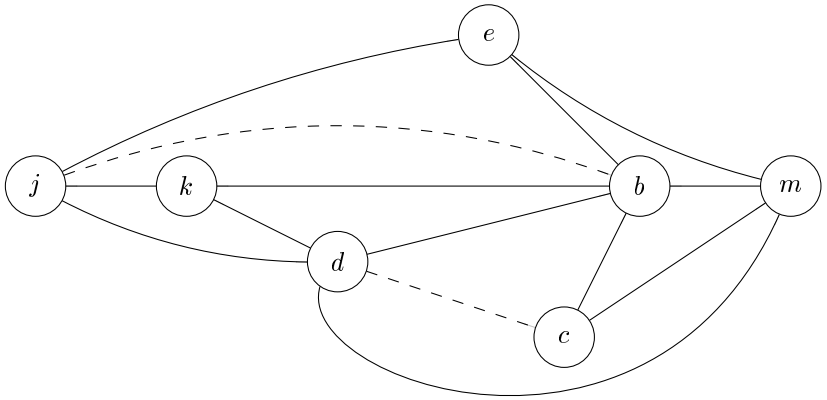
REMARQUE : Problème du constrained move : si `x<-y` et `y<-z`, avec une relation d'interférence entre x et z, on a le triangle :

[1] En anglais, to coalesce

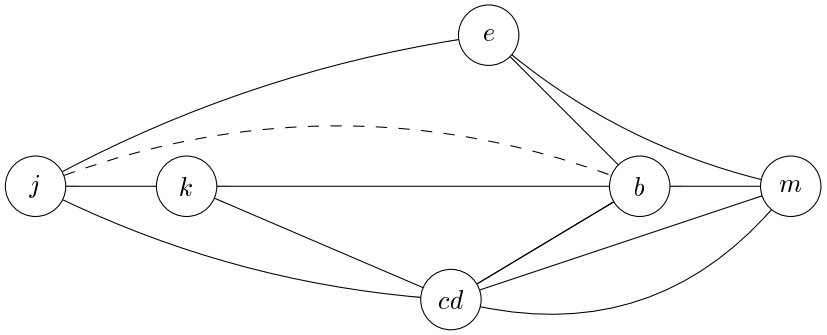


Alors, si on fusionne x et y on doit s'arrêter là car on ne peut plus fusionner z avec le nœud obtenu.

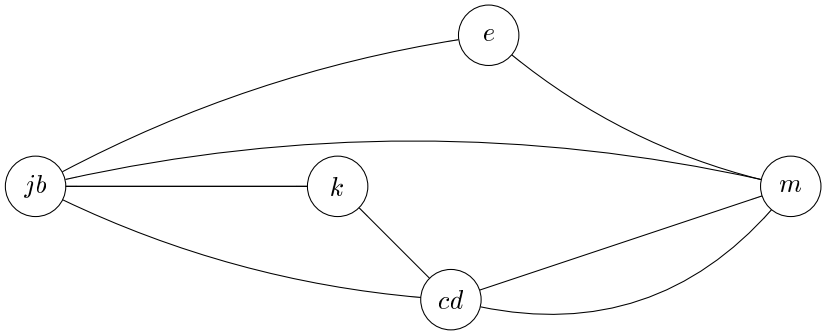
On va reprendre le même exemple, avec $K = 4$ registres/couleurs. Après avoir simplifié g, h et f, on se retrouve avec le graphe suivant :



On décide alors de fusionner les nœuds c et d : à eux deux, ils ont 4 voisins, mais k n'est pas significatif :



On fusionne encore j et b :



REMARQUE : Expérimentalement, on s'aperçoit que fusionner des nœuds au hasard au milieu de la procédure marche bien.

8.2.3 Noeuds précolorés

Mais tout n'est pas aussi facile que cela, par exemple il y a des nœuds précoloriés, à usage réservé, par exemple le FP, le premier argument ou le deuxième argument d'une procédure, le retour d'une fonction ...On ne doit ni les simplifier ni les "spiller" donc on leur accorde virtuellement un degré infini. On effectue alors la procédure précédente où l'on s'arrête lorsqu'il ne reste que des nœuds pré-colorés.

Notion de caller-save et callee-save :

Si la procédure f utilise une variable locale r, et qu'une sous-procédure g utilise aussi une variable locale r, il faut sauver la valeur locale du premier r avant l'appel de la procédure g. Si c'est la procédure f qui s'en charge, on dit que r est dans un registre **caller save** et si c'est g qui s'en charge, on dit que r va dans un registre **callee-save**.

Temporary copies :
Si $r7$ est un registre machine callee save à l'entrée d'une procédure, on le copie pour alléger la pression sur les registres :

```
enter [def(r7)]
    t_123=r7    copie de r7 dans t_123 registre virtuel
       ....
       ....     r7 est libre !
       ....
    r7=t_123
exit [use(r7)]
```

8.3 Traitement d'un exemple complet

On traite un exemple pour une machine à trois registres physiques r_1, r_2 et r_3. On suppose que r_1, r_2 sont des registres *caller save* et que le troisième registre r_3 est *callee save*. Notons que dans cet exemple à trois registres comme en général, la répartition entre les deux catégories *caller save* et *callee save* se décide ...lors de la conception du compilateur. Elle est spécifique à un couple compilateur/processeur donné.

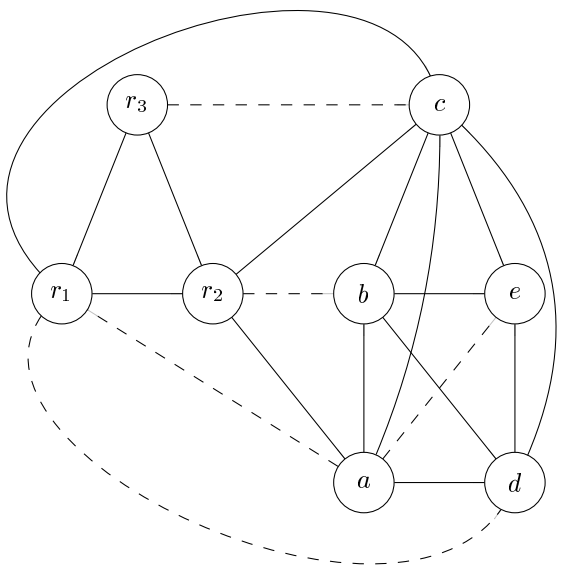
Soit le code suivant :

```
int f(int a, int b) {
     int d=0;
     int e=a;
     do  { d=d+b;
           e=e-1;
         } while (e>0);
     return d;
}
```

A l'aide des chapitres précédents, on génère facilement le code qui va suivre (avec pseudo-registres). Les arguments de la procédure sont passés via les registres physiques r_1, r_2, qui sont donc sont live-in en entrée de la procédure. Le troisième registre r_3 *callee save* est temporairement copié en entrée.

```
enter   c=r3;   //temporary copy
        a=r1;   //1er argument
        b=r2;   //2eme argument
        d=0;
        e=a;
   loop:d=d+b;
        e=e-1
        if e>0 goto loop
        r1<-d   //retour fonction
        r3<-c;  //restauration
return  on suppose (r1,r3) live_out
```

On construit le graphe d'interférence :



Tous les noeuds non précolorés ont un degré supérieur à $K = 3$. Il faut spiller une variable. Comment choisit-on la variable à spiller ? En introduisant une notion de spill priority :

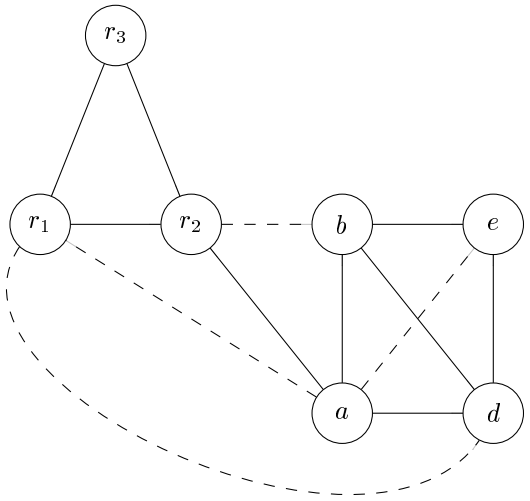
Node	Use/defs "out"	Use/defs "in"	Degré	Spill priority
a	2	0	4	0.50
b	1	1	4	2.75
c	2	0	6	0.33
d	2	2	4	5.50
e	1	3	3	10.33

La donnée caractéristique "Spill priority" est calculée en supposant qu'une boucle donnée est faite 10 fois en moyenne, d'où la formule "expérimentale" :

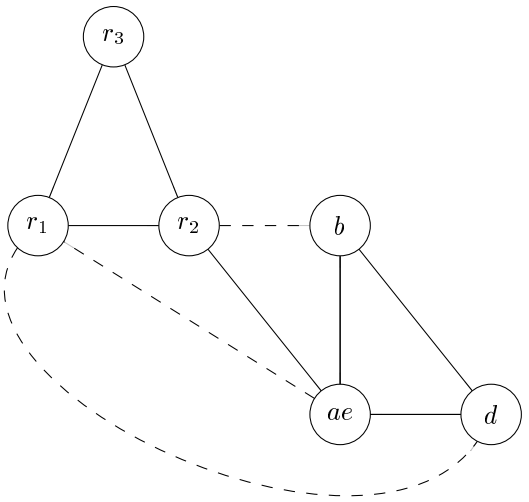
$$spill = (out + 10 * in)/degree.$$

L'idée est que plus on est profond dans une boucle, plus il faut mettre en registre car on utilise ladite variable plus souvent.

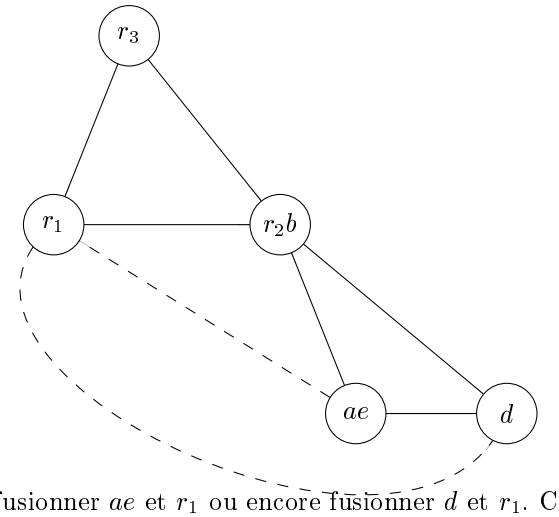
On choisit donc ici de spiller c en premier, ce qui donne le graphe :



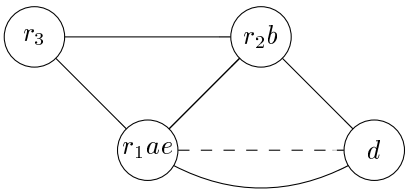
On peut fusionner a et e, car le noeud résultant ne sera adjacent qu'à deux noeuds significatifs : après la fusion, d sera de degré deux.



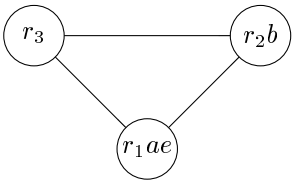
On peut alors fusionner ae et r_1 ou encore fusionner b et r_2. Choisissons cette dernière possibilité :



On peut alors fusionner ae et r_1 ou encore fusionner d et r_1. Choisissons la première possibilité :



Enfin, on ne peut fusionner (r_1ae) et d car le move est "constrained". On doit simplifier d. Il ne reste alors que des noeuds pré-colorés :

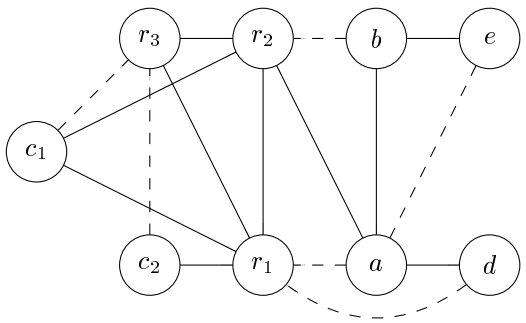


On dépile les variables : d reçoit la couleur r_3. La variable c, qui était marquée PS potential spill, est ne fait un vrai spill, on ne peut pas la colorier. Finalement, on obtient le code :

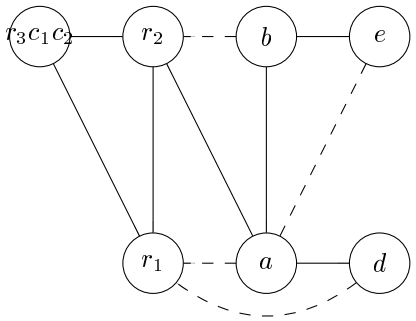
```
enter :  c1 :=r3;
         M[c_loc]:=c1;
         a:=r1;
         b:=r2;
         d:=0;
         e:=a;
    loop: d:=d+b;
         e:=e-1;
         if e>0 goto loop;
         r1:=d;
```

```
        c2:=M[c_loc];
        r3:=c2;
return
```

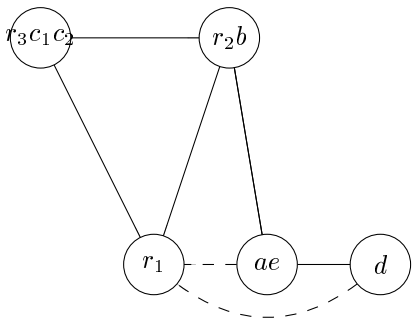
Et on recommence (à cause de l'introduction de c_1 et c_2) en reconstruisant un nouveau graphe d'interférence :



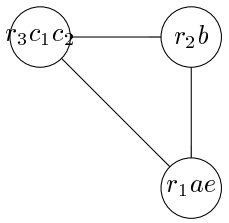
puis le début du coloriage : on fusionne immédiatement c_1 et r_3, puis c_2 et r_3 :



puis (fusion de ae et br_2) :



enfin (fusion de ae et r_1, simplification de d) :



puis le coloriage :

Nœud	Color
a	r1
b	r2
c	r3
d	r3
e	r1

Maintenant on peut réécrire le programme de la façon suivante :

```
enter :  r3 :=r3;
         M[c_loc]:=r3;
         r1:=r1;
         r2:=r2;
         r3:=0;
         r1:=r1;
   loop: r3:=r3+r2;
         r1:=r1-1;
         if r1>0 goto loop;
         r1:=r3;
         r3:=M[c_loc];
         r3:=r3;
return
```

puis enfin on peut enlever toute instruction qui est du type $reg := reg$, ce qui est était bien le but des opérations de fusion :

```
enter :  M[c_loc]:=r3;
         r3:=0;
   loop: r3:=r3+r2;
         r1:=r1-1;
         if r1>0 goto loop;
         r1:=r3;
         r3:=M[c_loc];
return
```

Le programme final a une seule instruction move qui n'a pas été fusionnée.

Chapitre 9

Analyse data flow

On a déjà fait de l'analyse data-flow dans le chapitre précédent, me dites-vous [1]. En effet, la "liveness analysis" est un cas particulier d'analyse data-flow. On va ici voir d'autres exemples, tout aussi utiles.

9.1 Reaching definitions

On ne regarde pas ce qui se passe en mémoire. Sinon, on ne fait pas de l'analyse data-flow, mais de l'alias-analysis.

On se limite ici aux pseudo-registres créés lors de la génération de code.

DÉFINITION 9.1.1

- $t \leftarrow a \oplus b$ est une définition d de t en un nœud n du CFG.
- Cette définition **atteint un sommet u** si il existe un chemin du CFG de n vers u sur lequel il n'y a aucune définition de t.
- Une instruction $t \leftarrow a \oplus b$ tue toutes les définitions précédentes de t.
- L'ensemble $def(t)$ est l'ensemble des numéros de statement où il existe une définition de t.
- $in(n)$ (resp. $out(n)$) est l'ensemble des variables qui atteignent l'entrée du nœud n (resp. la sortie).

DÉFINITION 9.1.2 (GEN ET KILL)
Si on a une instruction $s : t \leftarrow a \oplus b$ au nœud d, alors $gen(s) = d$ et $kill(s) = def(t) \setminus \{d\}$ d'après ce qu'on a dit précedemment.

Alors on a les relations entre ensembles :

$$in(n) = \bigcup_{p \in prec(n)} out(p)$$

et :

$$out(n) = gen(n) \cup (in(n) \setminus kill(n)),$$

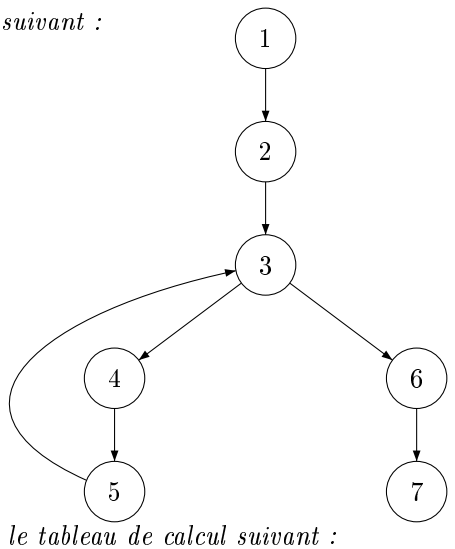
relations qui vont permettent de calculer ces ensembles par itération jusqu'à point fixe.

[1] C'est bien, vous suivez

EXEMPLE 9.1.1 *Soit le code suivant :*

```
1      a:=5;
2      c:=1;
3 L1:  if c> a goto L2
4      c:=c+c;
5      goto L1;
6 L2:  a:=c-a;
7      c:=0;
```

Dont le CFG est le suivant :



On obtient alors le tableau de calcul suivant :

state	*gen*	*kill*	*1-in*	*1-out*	*2-in*	*2-out*
1	*1*	*6*		*1*		*1*
2	*2*	*4,7*	*1*	*1,2*	*1*	*1,2*
3			*1,2*	*1,2*	*1,2,4*	*1,2,4*
4	*4*	*2,7*	*1,2*	*1,4*	*1,2,4*	*1,4*
5			*1,4*	*1,4*	*1,4*	*1,4*
6	*6*	*1*	*1,2*	*2,6*	*1,2,4*	*2,4,6*
7	*7*	*2,4*	*2,6*	*6,7*	*2,4,6*	*6,7*

REMARQUE : A quoi ça sert ? A la ligne 3 on sait qu'il n'y a qu'une seule définition de a. On peut alors faire une propagation de constante car on est sur que a vaut toujours 5.

9.2 Available expressions

DÉFINITION 9.2.1
$x \oplus y$ est **disponible** au nœud n si quelque soit le chemin qui mène de S_0 à n, $x \oplus y$ est calculé au moins une fois et si ni x ni y n'ont été modifié depuis la dernière occurrence du calcul.

DÉFINITION 9.2.2
A la base, $s : t \leftarrow a \oplus b$ est une **génération** de $a \oplus b$. Mais ce n'est vrai que si Si $t \neq a$ et $t \neq b$. Alors $gen(s) = \{a \oplus b\} \setminus kill(s)$, où $kill(s)$ est l'ensemble des expressions qui contiennent t.

On obtient le tableau suivant :

Statement s	gen[s]	kill[s]
$t \leftarrow a \oplus b$	$\{a \oplus b\}$ - kill[s]	expressions contenant t
$t \leftarrow M[b]$	$\{a \oplus b\}$ - kill[s]	expressions contenant t
$M[a] \leftarrow b$	$\{\}$	expressions de la forme $M[x]$

Noter qu'une instruction store $M[a] \leftarrow b$ peut modifier n'importe qu'elle case mémoire a priori, donc elle tue toute les instructions $M[x]$ qui adressent la mémoire. Si on avait des informations supplémentaires garantissant que a et x ne pointent pas au même emplacement, on pourrait faire mieux : c'est l'analyse d'alias.

On a ici les relations de récurrence suivantes :

$$in(n) = \bigcap_{p \in prec(n)} out(p)$$

et :

$$out(n) = gen(n) \cup (in(n) \setminus kill(n)),$$

relations qui vont ici aussi permettre de calculer ces ensembles par itération jusqu'à point fixe.
REMARQUE : Remarquez ici l'intersection pour la définition de in. En effet, on quantifie ici sur *tous* les chemins.

Pour l'initialisation, on fait $in(entree) \leftarrow \emptyset$ et $in(autres) =$ toutes les variables. Ici par contre on trouve par itération le plus grand point fixe.

DÉFINITION 9.2.3
Sont "reaching expressions" en s les statements $n : v \leftarrow x \oplus y$ telles que le chemin de n à s ne redéfinisse ni x ni y ni ne recalcule $x \oplus y$.

On peut annoter le calcul des expressions disponibles pour avoir ces expressions qui atteignent s.

9.3 Optimisation avec l'analyse data-flow

On va utiliser ici ce que l'on a fait dans les sections précédentes afin d'optimiser le code obtenu.

- **Sous-expressions communes**
  Soit $s : t \leftarrow x \oplus y$ où $x \oplus y$ est disponible en s. On élimine le re-calcul en s en faisant :
  1. Calculer les "reaching expressions", *i.e.* les $n : s \leftarrow x \oplus y$ qui sont à l'origine du calcul (noter qu'on peut avoir plusieurs branches dans le CFG, et donc plusieurs n)
  2. On prend un nouveau temporaire, par exemple w.
  3. On remplace toutes les reaching expressions obtenues en 1 par $n : w \leftarrow x \oplus y$ et (sauvegarde) $n' : v \leftarrow w$
  4. On remplace s par $s : t \leftarrow v$.

- **Propagation de constante**
  Soit un statement $s : d : t \leftarrow c$ où c est une constante. Alors soit n un autre statement qui utilise t, par exemple $n : y \leftarrow t \oplus x$. Alors si d atteint n et aucune autre définition de t n'atteint n, on peut remplacer $n : c \oplus x$.

- **Copy popagation**
  Pour un statement s est $d : t \leftarrow z$ et un autre : $n : y \leftarrow t \oplus x$, avec z variable, si d atteint n, si aucune définition de z n'a lieu sur un chemin de s à n, alors on peut remplacer $n : y \leftarrow z \oplus x$.

- **Dead-code elimination**
  Si $s : a \leftarrow b \oplus c$ ou $a \leftarrow M[x]$ avec a non live-out en s, alors on peut détruire s.

  REMARQUE : Il faut être vigilant aux effets de bords de l'instruction s. Si par hasard elle renvoie une exception arithmétique (division par 0, *etc.*), il y a un problème !

9.4 Accélération

Pourquoi veut-on accélérer l'analyse data flow ? En fait, le nombre d'itérations au pire est très grand ($O(n^4)$, et donc on peut mouliner un moment !

9.4.1 Au niveau des basic blocks

Jusqu'à présent, on a tout fait au niveau des expressions. Mais souvent il est préférable vu le nombre de ces expressions et la complexité de l'analyse data-flow, de faire cette même analyse au niveau plus macroscopique des basic blocks. On utilise la proposition suivante :

PROPOSITION 9.4.1 *Si p est le seul prédécesseur de n dans le graphe (ce qui est le cas si p et n sont dans un basic-block, par définition), alors on peut fusionner les nœuds p et n et :*

$$gen(pn = gen(n) \cup (gen(p) \setminus kill(p))$$

$$kill(pn) = kill(p) \cup kill(n)$$

Ainsi, si on veut connaître les d'informations au niveau des blocs de base, on peut calculer les mêmes caractéristiques data-flow que précédemment, mais sur moins de nœuds. Il y a moins d'informations, mais c'est plus rapide et cela peut quelquefois suffire.

9.4.2 Ordre des nœuds

L'ordre de parcours change suivant l'analyse : le parcours en profondeur correspond à la "forward analysis" et le parcours arrière à la "backward analysis". Pour les problèmes arrières (par exemple la liveness analysis), on part de l'exit node au lieu de l'entry node.

REMARQUE : Nœuds de début et de fin n'ont pas vraiment de sens car le graphe qur lequel on travaille peut être cyclique. Ceci dit, on peut dire "avec les mains" qu'il y a un sens privilégié de parcours.

9.4.3 Value numbering

Voici un exemple d'algorithme "brute-force" où en une seule passe on trouve toutes les sous-expressions communes :

```
g:=x+y
h:=u-v
i:=x+y
x:=u-v
u:=g+h
v:=i+x
w:=u+v
```

On maintient à jour une table de hachage T où l'on rentre :
- Les variables
- Les triplets (x, op, y)

L'accès à ce tableau devra être rapide pour savoir si une sous-expression a déjà été calculée.

On remplit donc successivement :

$$\begin{cases} x & \mapsto 1 \\ y & \mapsto 2 \\ (1,+,2) & \mapsto 3 \\ g & \mapsto 3 \\ u & \mapsto 4 \\ v & \mapsto 5 \\ (4,-,5) & \mapsto 6 \\ h & \mapsto 6 \\ i & \mapsto 3 \\ x & \mapsto 6 \\ (3,+,6) & \mapsto 7 \\ u & \mapsto 7 \\ v & \mapsto 7 \\ (1,+,7) & \mapsto 8 \\ w & \mapsto 8 \end{cases}$$

REMARQUE : Evidemment, une sur-définition d'une expression écrase la précédente !

9.5 Alias analysis

On survole en cours deux méthodes, l'une liée au système de typage, et l'autre liée à la partition de la mémoire en classes d'alias potentiels.

On renvoie au livre d'Appel pour plus de précisions ...

Chapitre 10

Références

Nous avons utilisé principalement deux livres pour ce cours :

Modern compiler design de Dick Grune, Henri E. Bal, Ceriel J.H. Jacobs, et Koen G. Langendoen. Pour toute la partie front-end, et pour la génération de code (avec en particulier le BURS pour la sélection des instructions).

Modern Compiler Implementation in Java de Andrew W. Appel (Cambridge University Press). Essentiellement pour les optimisations : allocation de registres et analyse data-flow.

Ces deux ouvrages récents ont vigoureusement rajeuni le domaine. Saluons pour mémoire le *dragon book*, auquel il est toujours bon de revenir : *Compilers : Principles, Techniques, and Tools* de Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman (Addison-Wesley, traduction InterEditions).

Plusieurs exemples du cours ont été tirés du livre : *Programming Language Pragmatics*, de Michael L. Scott (Morgan Kaufmann). Ce livre très riche étudie la compilation des langages à partir de leur diversité : il en traite un grand nombre dont Algol, Fortran, Pascal, C, Java, Scheme, ML, Lisp, Prolog, ...

Enfin, la bible des optimisations, *the definitive book on compiler optimization*, est le livre : *Advanced Compiler Design and Implementation*, de Steven S. Muchnick. Indispensable pour tout professionnel de la compilation en devenir :-)