

# Cours de Compilation, Master 1, 2004-2005

Tanguy Risset surtout, Florent de Dinechin un peu

3 janvier 2006

## 1 Généralités

Page web du cours :

<http://perso.ens-lyon.fr/florent.de.dinechin/enseignement/compil/>

### 1.1 Biblio

La construction d'un compilateur rassemble une grande diversité de processus. Certains sont une application directe de résultats théoriques puissants (comme la génération automatique d'analyseurs lexicaux qui utilise la théorie des langages réguliers). D'autres se rattachent à des problèmes difficiles, NP-Complet (ordonnancement d'instruction, allocation de registres).

Ce qui est probablement le plus délicat dans la construction d'un compilateur, c'est qu'il faut trouver un équilibre entre un code produit efficace et un temps de production de code raisonnable.

Il est très important de réaliser que la maîtrise des techniques de construction d'un compilateur est l'une des raisons qui rendent les informaticiens indispensables. Un biologiste peut savoir bien programmer, mais lorsqu'il aura à faire des traductions entre différents formats, il va utiliser des méthodes ad-hoc. L'informaticien peut apporter énormément à une équipe par la maîtrise de cet outil très puissant qu'est le compilateur.

Le but de ce cours est de faire assimiler les techniques de bases de compilation pour

1. pouvoir rapidement écrire un compilateur : utiliser les techniques et outils ayant fait leurs preuves.
2. avoir assimilé les notions qui permettent de comprendre les problèmes et les solutions dans les compilateurs modernes.

Ce cours a été préparé à partir du livre de Keith D. Cooper et Linda Torczon de Rice University : "Engineering a Compiler" [CT03]. Il est aussi issu du cours de compilation fait par Yves Robert jusqu'en 2003 et Tanguy Risset en 2004. Parmi les ouvrages importants qui peuvent compléter ce cours je recommande [CT03, ASU88, Muc]. D'autre part, une bonne connaissance de l'architecture des processeurs peut être utile si l'on décide d'approfondir un peu la compilation. La référence la plus accessible est le livre de Hennessy et Patterson [HP98]

## Références

[ASU88] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : Principles, Techniques and Tools*. Addison-Wesley, 1988. La bible, un peu dépassé sur certains sujets, mais beaucoup n'ont pas changé depuis.

[CT03] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Morgan-Kaufmann, 2003. Le livre utilisé pour ce cours, en cours de commande à l'ENS.

[Muc] Steven S. Muchnick. *Compiler Design Implementation*. Morgan-Kaufmann. Très complet pour approfondir les optimisations.

[wA98] Andrew w. Appel. *Modern Compiler implementation in Java*. Cambridge University press, 1998. Certaines parties très bien expliquées

- [GBJL00] D. Grune, H. Bal, J. H. Jacobs, and K. Langendoen. *Modern Compiler Design*. John Wiley & Sons, 2000. Globalement un peu moins bien (sauf pour certains points précis).
- [HP98] J. Hennessy, D. Patterson *Computer Organization and Design : The hardware software interface* Morgan Kaufman 1998. Référence pour la description des architectures Risc.

## 1.2 Qu'est ce qu'un compilateur ?

**Définition 1** *Un compilateur est un programme qui prend en entrée un programme exécutable et produit en sortie un autre programme exécutable.* □

Le langage du programme source est appelé langage source, celui du programme cible est appelé langage cible. Exemple de langage sources : Fortran, C, Java etc... Le langage cible peut être un de ces langages ou un assembleur ou un code machine pour une machine précise.

Beaucoup de compilateurs ont le langage C comme langage cible car il existe des compilateurs C sur pratiquement toutes les plate-formes. Un programme qui génère du postscript à partir d'un format donnée (ascii, latex, jpg, etc.) est aussi un compilateur. En revanche, un programme qui transforme du postscript en des pixel sur un écran est un *interpréteur*.



Exemple de langage interprété : Perl, Scheme, Mathematica, Maple.

Les principes fondamentaux de la compilation sont :

1. Le compilateur doit conserver le sens du programme compilé
2. Le compilateur doit améliorer le code

les propriétés importantes d'un compilateur sont :

- code produit efficace (rapidité, mémoire).
- informations retournées en cas d'erreurs, déverminage
- rapidité de la compilation

Dans les années 80, les compilateurs étaient tous de gros systèmes monolithiques générant du code assembleur le plus rapide possible directement à partir d'un programme entier. Aujourd'hui d'autres fonctions de coût entrent en jeu pour juger de la qualité d'un compilateur : la taille de code et la consommation électrique ont une importance primordiale pour les systèmes embarqués (pour les applications multimédia récentes, la taille de code est quasiment le critère numéro 1).

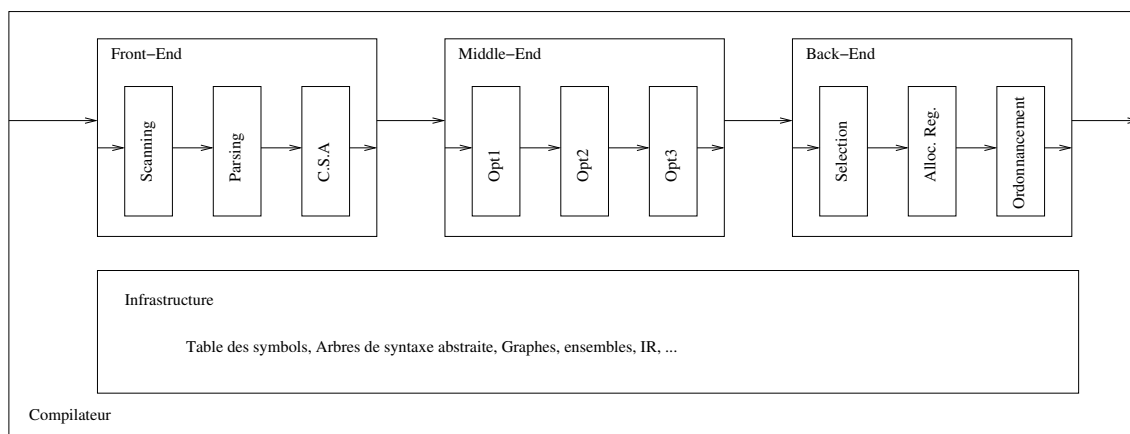
Le processus de compilation peut aussi être décomposé : par exemple le *bytecode* java est un programme *semi-compilé*. L'éditeur de liens, reliant le programme compilé avec les bibliothèques précompilées qu'il utilise peut éventuellement faire des optimisations. Le programme peut même être amélioré à l'exécution avant d'être exécuté grâce à des informations supplémentaires présentes à l'exécution.

## 1.3 Un exemple simple

Considérons l'expression suivante pour laquelle nous voulons générer du code exécutable :

$$w \leftarrow w \times 2 \times x \times y \times z$$

De nos jours, on décompose un compilateur en 3 *passes* (front-end, middle-end, back-end) bien qu'il y ait beaucoup plus de trois passes en pratique. En plus des actions de chaque passe, il est important de définir précisément les *représentations intermédiaires* (*intermediate representation*, IR) utilisées ainsi que l'infrastructure du compilateur pour manipuler ces représentations. Une représentation intuitive est la suivante :



On appelle cela un compilateur optimisant (optimizing compiler) par opposition aux premiers compilateurs qui ne comprenaient que deux passes (front-end, back end).

### 1.3.1 front-end

La première tâche du compilateur est de comprendre l'expression du langage source. Cela se fait en plusieurs étapes :

1. Analyse syntaxique : permet de vérifier que l'expression est bien une phrase du langage source syntaxiquement correcte, on dit aussi qu'elle est *bien formée*. Cela nécessite donc une définition formelle du langage source. Exemple en français : *Le lion mange de la viande* est syntaxiquement correcte et *le lion viande* n'est pas syntaxiquement correcte. En pratique, cette phase est divisée en deux traitements : l'analyse lexicale ou *scanning* (repérer les césures de mots, la ponctuation) et l'analyse syntaxique ou *parsing* (vérifier les règles de grammaire pour l'exemple du français).
2. Analyse sémantique : permet de vérifier que l'expression a un sens dans le langage source (on peut dire aussi analyse sensible au contexte, *context sensitive analysis*, CSC en anglais). Cela nécessite une sémantique précise pour le langage source. Exemple en Français : *le lion dort de la viande* est syntaxiquement correcte (sujet, verbe, complément d'objet) mais n'a pas de sens défini. Ici, on peut être amené à se demander si les variables  $w, x, y$  et  $z$  ont été déclarées, et si on leur a affecté des valeurs précédemment.

Ces traitements sont regroupés dans ce qui est appelé le *front-end* du compilateur. Ces deux traitements sont largement automatisés aujourd'hui grâce à l'application des résultats de la théorie des langages. On dispose d'outils permettant de générer les analyseurs lexicaux et syntaxiques à partir de la description du langage source sous forme de grammaire.

### 1.3.2 Création de l'environnement d'exécution

Le langage source est généralement une abstraction permettant d'exprimer un calcul dans un formalisme relativement intuitif pour l'être humain. Par exemple, il contient des noms symboliques :  $x, y, z, w$  qui représente plus que des valeurs : des cases mémoire qui peuvent prendre plusieurs valeurs successivement.

Le compilateur doit détruire cette abstraction, faire un choix pour implémenter la structure et les actions désignées et maintenir la cohérence avec les actions qui suivent. Ici, par exemple le compilateur peut choisir d'allouer quatre cases mémoire consécutives pour les quatre variables  $w, x, y, z$  :

$${}^0 \begin{array}{|c|c|c|c|} \hline w & x & y & z \\ \hline \end{array} ,$$

ou il peut décider de conserver ces variables dans des registres par une série d'affectation :

$$r_1 \leftarrow w; r_2 \leftarrow x; r_3 \leftarrow y; r_4 \leftarrow z$$

Dans tous les cas, le compilateur doit assurer la cohérence de ces choix tout au long du processus de traduction.

En plus des noms le compilateur doit créer et maintenir des mécanismes pour les procédures, les paramètres, les portées lexicales les opérations de contrôle du flot. On appelle cela choisir la *forme* du code.

Ce point précis est assez important : c'est parce que l'on a étudié le processus de compilation vers une architecture cible d'un type très précis : type architecture de Von Neumann séquentielle avec un processeur programmable (muni de plusieurs unités) et une mémoire (éventuellement hiérarchisée) que l'on a pu automatiser cette *descente* dans le niveau d'abstraction. Aujourd'hui, il n'existe toujours pas de paralléliseur satisfaisant pour une machine parallèle donnée (c'est d'ailleurs une des raisons pour laquelle les constructeurs ne fabriquent quasiment plus de machines parallèles) car le *raffinement* efficace de l'abstraction pour ces architectures est beaucoup plus délicat (les communications posent un énorme problème de performance). De même on ne sait pas compiler efficacement un circuit intégré à partir d'une spécification de haut niveau car pour beaucoup d'abstraction faites par les langages de haut niveau les choix à faire pour l'implémentation sont très difficiles. Par exemple l'introduction du temps à partir d'une spécification fonctionnelle (ordonnancement), la parallélisation etc.

### 1.3.3 Améliorer le code

Le compilateur peut très souvent tirer partie du contexte dans lequel se trouve une expression pour optimiser son implémentation. Par exemple, si l'expression à compiler se trouve à l'intérieur d'une boucle dans laquelle la sous expression  $2 \times x \times y$  est invariante, le compilateur aura intérêt à introduire une variable temporaire pour sortir le calcul de cette expression de la boucle :

```
w <- 1
for i=1 to n
  read z
  w <- w * 2 * x * y * z
end

w <- 1
t1 <- 2 * x * y
for i=1 to n
  read z
  w <- w * t1 * z
end
```

Cette phase de la compilation est généralement séparée en des passes d'*analyse* et des passes de *transformation*. Les analyses vont permettre de mettre en place les objets utilisés lors des transformations. Cette phase est quelque fois appelée *middle-end*.

L'*analyse data flow* permet de raisonner au moment de la compilation sur la manière dont les valeurs seront transmises à l'exécution. L'analyse de dépendance sert à rendre non ambiguës les références à des éléments de tableaux.

Nous verrons quelques transformations mais il en existe énormément, toutes ne peuvent être mentionnées ici.

### 1.3.4 La génération de code

C'est la partie qui diffère d'avec les interpréteurs. Le compilateur traverse la structure de données qui représente le code et émet un code équivalent dans le langage cible. Il doit choisir les instructions à utiliser pour implémenter chaque opération (sélection d'instruction), décider quand et où copier les valeurs entre les registres et la mémoire (allocation de registre), choisir un ordre d'exécution pour les instructions choisies (ordonnancement d'instructions). Souvent, les optimisations de ces différentes phases entrent en conflit. Cette phase est aussi appelé *back end*.

Deux manières de générer le code pour l'expression

$$w \leftarrow w \times 2 \times x \times y \times z$$

```

loadAI  r_arp, @w    => r_w    // load w
loadI   2            => r_2    // la constante 2 dans r_2
loadAI  r_arp, @x    => r_x    // load x
loadAI  r_arp, @y    => r_y    // load y
loadAI  r_arp, @z    => r_z    // load z
mult    r_w, r_2     => r_w    // r_w ← w×2
mult    r_w, r_x     => r_w    // r_w ← (w×2)× x
mult    r_w, r_y     => r_w    // r_w ← (w×2)× x × y
mult    r_w, r_z     => r_w    // r_w ← (w×2)× x × y × z
storeAI r_w          => r_arp, @w // écriture de w

```

Le code ci dessous utilise deux registres au lieu de 5 :

```

loadAI  r_arp, @w    => r_1    // load w
add     r_1, r_1     => r_1    // r_1 ← w× 2
loadAI  r_arp, @x    => r_2    // load x
mult    r_1, r_2     => r_1    // r_1 ← (w×2)× x
loadAI  r_arp, @y    => r_2    // load y
mult    r_1, r_2     => r_1    // r_w ← (w×2)× x × y
loadAI  r_arp, @z    => r_2    // load z
mult    r_1, r_2     => r_1    // r_1 ← (w×2)× x × y × z
storeAI r_1          => r_arp, @w // écriture de w

```

## 1.4 assembleur Iloc

Un compilateur est étroitement lié à l'assembleur qu'il va générer (au moins pour les phases du back-end). Il existe autant d'assembleur que de type de processeurs, cependant on peut dégager des principes communs. Pour illustrer le cours on utilisera le langage assembleur Iloc proposé dans [CT03]. ILOC est un code lineaire assembleur pour une machine RISC. C'est une version simplifiée de la représentation intermédiaire utilisée dans le MSCP (Massively Scalar Compiler Project), développé à l'université de Rice. Iloc est détaillé en Annexe A (page 111). Des exemples d'assembleur réel (Pentium et MIPS) sont visibles en pages 130 et 131.

ILOC n'a qu'un type de nombre : les entiers (pas de double, flottant etc.). La machine abstraite sur laquelle ILOC s'exécute possède un nombre illimité de registres. C'est un code trois adresses. Il supporte simplement les modes d'adressage élémentaires :

- adressage direct.
  - load r<sub>1</sub> => r<sub>2</sub> : charger dans r<sub>2</sub> le contenu de la case dont l'adresse est contenue dans r<sub>1</sub>,
- chargement.
  - loadI 2 => r<sub>2</sub> : charger dans r<sub>2</sub> la valeur 2.
- adressage indirect *Address-Immediate*.
  - loadAI r<sub>1</sub>, 2 => r<sub>2</sub> : charger dans r<sub>2</sub> le contenu de la case dont l'adresse est "contenu de r<sub>1</sub>+2".
- adressage indirect indexé *Adress-Offset*.
  - loadAO r<sub>1</sub>, r<sub>2</sub> => r<sub>3</sub> : charger dans r<sub>3</sub> le contenu de la case dont l'adresse est "contenu de r<sub>1</sub>+contenu de r<sub>2</sub>".<sup>1</sup>

Chaque instruction ILOC peut être précédée d'un label qui peut être référencé par les instructions de branchement (jump, cbr). Le registre r<sub>0</sub> désigne par défaut l'enregistrement d'activation (activation record : AR) de la procédure courante. Les décalage en mémoire par rapport à cet AR sont notés : @x. Il y a deux types d'instruction de contrôle du flôt pour illustrer deux manières d'implémenter cela au niveau de la génération de code.

---

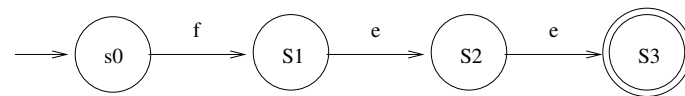
<sup>1</sup>demandeur le nom des différents adressage en francais

## 2 Grammaires et expressions régulières

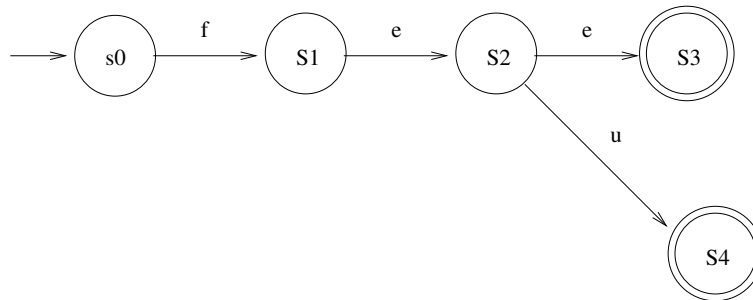
### 2.1 Quelques définitions

Nous allons survoler quelques définitions importantes (langage régulier, grammaire, automate etc.). Ces notions sont utilisées dans beaucoup de domaines de l'informatique, elles sont approfondies dans le cours de Marianne Delorme ("automates").

**automates finis** On cherche à reconnaître un langage c'est à dire à indiquer si la phrase que l'on a lu appartient bien syntaxiquement au langage ou pas. Commençons par la tâche simple qui consiste à reconnaître quelques mots particuliers. Par exemple, nous souhaitons reconnaître le mot "fee". La manière la plus naturelle est de lire les lettres les unes après les autres et d'*accepter* le mot une fois que l'on a bien lu "fee". On passe alors par quatre *états* : lorsqu'on a rien lu, lorsqu'on a lu 'f', lorsqu'on a lu 'fe' puis lorsqu'on a lu 'fee'. On peut représenter cela par le diagramme :



Sur ce dessin, l'état initial est  $s_0$ ,  $s_3$  est l'état final signifiant qu'on a reconnu le mot 'fee'. Si on est en train de lire un autre mot, une des transitions ne pourra pas s'effectuer, on rejettera le mot. Le diagramme ci-dessus est la représentation d'un automate fini qui reconnaît le mot 'fee'. Si l'on veut reconnaître deux mots, par exemple 'fee' et 'feu', on peut utiliser l'automate suivant :



Plus formellement, on peut définir la notion d'automate fini :

**Définition 2** Un automate fini déterministe est donné par  $(S, \Sigma, \delta, s_0, S_F)$  ou :

- $S$  est un ensemble fini d'états ;
- $\Sigma$  est un alphabet fini ;
- $\delta : S \times \Sigma \rightarrow S$  est la fonction de transition ;
- $s_0$  est l'état initial ;
- $S_F$  est l'ensemble des états finaux

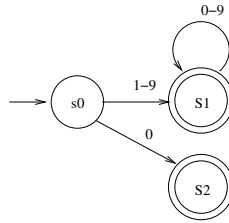
□

Dans notre exemple de l'automate reconnaissant "fee", on a  $S = \{s_0, s_1, s_2, s_3\}$ ,  $\Sigma = \{f, e\}$  (ou tout l'alphabet)  $\delta = \{\delta(s_0, f) \rightarrow s_1, \delta(s_1, e) \rightarrow s_2, \delta(s_2, e) \rightarrow s_3\}$ . Il y a en fait un état implicite d'erreur ( $s_e$ ) vers lequel vont toutes les transitions qui ne sont pas définies.

Un automate *accepte* une chaîne de caractère si et seulement si en démarrant de  $s_0$ , les caractères successifs de la chaîne entraînent une succession de transitions qui amène l'automate dans un état final. Si la chaîne  $x$  est composée des caractères  $x_1x_2 \dots x_n$ , on doit avoir :

$$\delta(\delta(\dots \delta(\delta(s_0, x_1), x_2), x_3) \dots, x_{n-1}), x_n) \in S_F$$

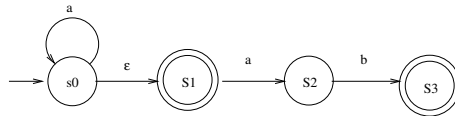
Par définition, on peut avoir des circuits dans le diagramme de transition d'un automate, c'est à dire que l'on peut reconnaître des mots de longueur arbitraire, par exemple, l'automate suivant reconnaît n'importe quel entier :



Les automates que nous avons vus jusqu'à présent sont tous déterministes, c'est à dire que lorsqu'ils sont dans un état particulier  $s_i$  et qu'ils lisent un caractère  $c_i$  la donnée  $s_j = \delta(s_i, c_i)$  définit exactement l'état suivant. Nous allons avoir besoin d'automates indéterministes, c'est à dire d'automates qui, étant dans un état  $s_i$  et lisant un caractère  $c_i$  pourront atteindre plusieurs états :  $s_{j_1}$ , ou  $s_{j_2}$ , ..., ou  $s_{j_3}$ . Il y a deux manières de représenter cela sur les diagrammes de transition : soit on autorise les  $\epsilon$ -transitions, c'est à dire des transitions du type :  $\delta(s_i, \epsilon) = s_j$  où  $\epsilon$  est la chaîne vide (i.e. l'automate peut changer d'état sans lire de caractère), soit on utilise plusieurs transitions étiquetées par un même caractère à partir d'un même état (c'est à dire que  $\delta$  devient une fonction de  $S \times \Sigma \rightarrow 2^S$ , qui retourne un ensemble d'états). En pratique, on fait les deux.

Nous avons alors besoin d'une nouvelle définition du fonctionnement d'un automate (ou de la notion d'accepter). Un automate non déterministe  $(S, \Sigma, \delta, s_0, S_F)$  accepte une chaîne de caractère  $x_0x_1 \dots x_n$  si et seulement si il existe un chemin dans le diagramme de transition qui démarre à  $s_0$ , et finit à  $s_k \in S_F$  et tel que les étiquettes des arcs épellent toute la chaîne (sachant que les arcs étiquetés par  $\epsilon$  ne comptent pas).

Par exemple : l'automate suivant reconnaît un mot d'un nombre quelconque (éventuellement nul) de 'a' ou un mot d'un nombre quelconque strictement positif de a suivi d'un 'b'.



On peut démontrer que les automates déterministes et non déterministes sont équivalents en terme de langage reconnus (on verra sur un exemple comment déterminer un automate).

**expressions régulières** Une expression régulière est une formule close permettant de désigner un ensemble de chaînes de caractères construites à partir d'un alphabet  $\Sigma$  (éventuellement augmenté de la chaîne vide  $\epsilon$ ). On appelle cet ensemble de chaîne de caractère un *langage*.

Une expression régulière est construite à partir des lettres de l'alphabet (un lettre dans une expression régulière désigne l'ensemble réduit à la chaîne de caractère égale au caractère en question) et des trois opérations de base sur les ensembles de chaînes de caractères :

- l'union de deux ensembles, notée  $R \mid S$  est  $\{s \mid s \in R \text{ ou } s \in S\}$ .
- la concaténation de deux ensembles  $R$  et  $S$  notée  $RS$  est  $\{rs \mid r \in R \text{ et } s \in S\}$ . On note  $R^i$  pour  $RRR \dots R$   $i$  fois.
- la fermeture transitive (ou étoile, ou fermeture de Kleene) d'un ensemble de chaînes de caractères  $R$  notée  $R^*$  est  $\cup_{i=0}^{\infty} R^i$ . C'est donc l'union des concaténation de  $R$  avec lui même zero fois (c'est à dire la chaîne vide  $\epsilon$ ) ou plus.

Le langage (ou ensemble) représenté par une expression régulière  $r$  est noté  $L(r)$ . On utilise aussi les parenthèses dans les expressions régulières pour exprimer la précedence (priorité de précedence : fermeture > concaténation > union). Enfin, on utilise des raccourcis de notation : par exemple  $[0..4]$  équivaut à  $0 \mid 1 \mid 2 \mid 3 \mid 4$ , c'est à dire l'ensemble constitué des cinq chaînes  $\{ '0', '1', '2', '3', '4' \}$ .

On utilise ce formalisme pour spécifier les langages de programmation que nous allons compiler. Par exemple, les identificateurs dans beaucoup de langages actuels sont définis comme commençant par un lettre suivi de lettres ou de chiffres. On peut décrire cet ensemble de chaînes par l'expression régulière :  $[a..z]([a..z] \mid [0..9])^*$ .

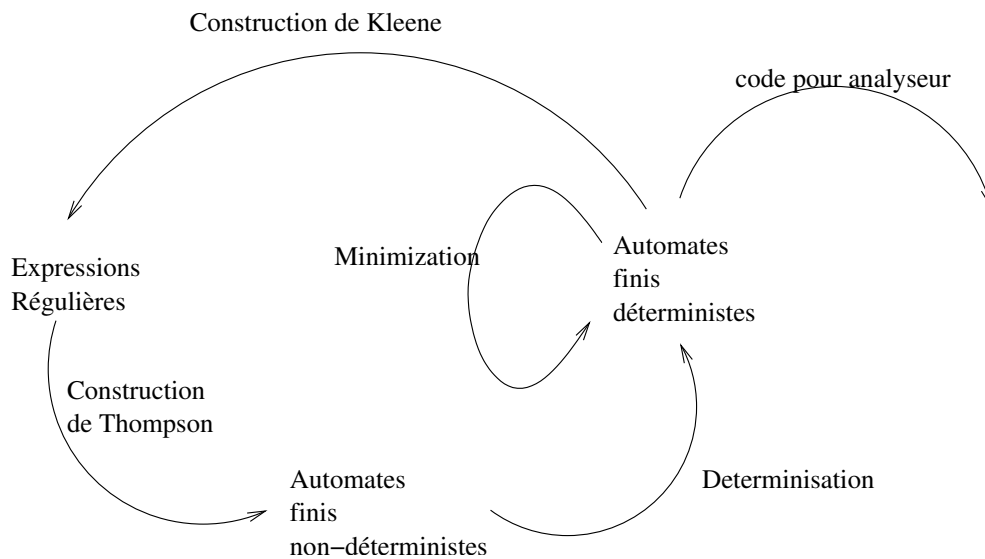
Pour les entiers :  $0 \mid [1..9][0..9]^*$

Pour les réels  $(+ \mid - \mid \epsilon)(0 \mid [1..9][0..9]^*)(.[0..9]^* \mid \epsilon)E(+ \mid - \mid \epsilon)(0 \mid [1..9][0..9]^*)$

L'ensemble des langages qui peuvent être exprimés par des expressions régulières est appelé l'ensemble des *langages réguliers*. Ces langages ont de bonnes propriétés qui les rendent en particulier aptes à être reconnus par des analyseurs générés automatiquement. Une propriété importante est que ces langages sont clos par les opérations d'union, concaténation et fermeture de Kleene ce qui permet de construire des analyseurs incrémentalement. On peut montrer (on va le voir sur un exemple) que les langages réguliers sont exactement les langages qui peuvent être reconnus par un automate fini.

## 2.2 De l'expression régulière à l'automate minimal

La théorie des langages réguliers permet de générer automatique un analyseur syntaxique à partir d'une expression régulière. Le schéma global est le suivant :

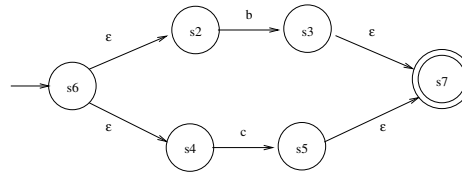
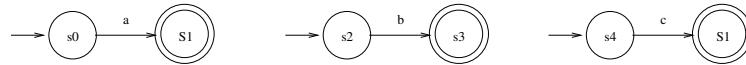


### 2.2.1 Des expressions régulières aux automates non déterministes

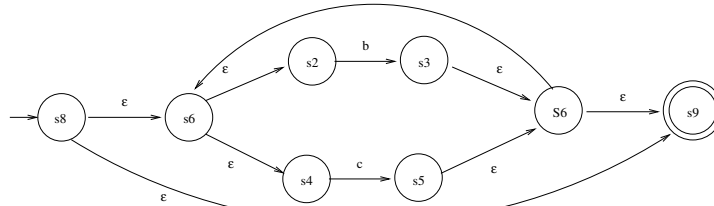
Pour être capable de construire un automate qui reconnaisse un langage régulier quelconque, il suffit d'avoir un mécanisme qui reconnaît chaque lettre, puis un mécanisme qui reconnaît la concaténation, l'union et la fermeture de langages reconnus.

Nous allons illustrer ce processus sur un exemple simple, il est élémentaire à généraliser. Considérons le langage désigné par l'expression régulière :  $a(b | c)^*$ . Le parenthésage indique l'ordre dans lequel on doit décomposer l'expression ; il faut reconnaître dans l'ordre : les langages  $b$  et  $c$  puis  $(b | c)$  puis  $(b|c)^*$  puis  $a$  puis  $a(b|c)^*$ .

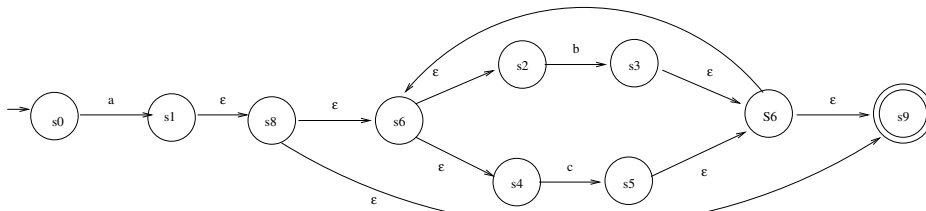




NFA pour  $b|c$

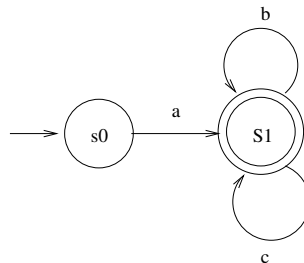


NFA pour  $(b|c)^*$



NFA pour  $a(b|c)^*$

Notons que ce langage particulier aurait été reconnu par un automate beaucoup plus simple :



La suite du processus permettra de simplifier l'automate obtenu jusqu'à obtenir l'automate ci-dessus.

## 2.2.2 Déterminisation

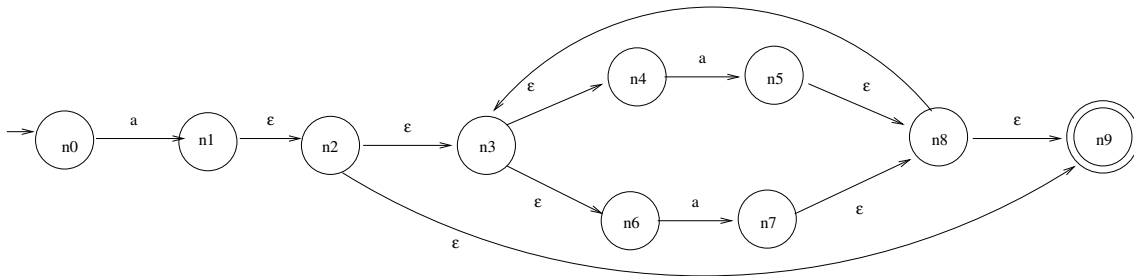
On veut maintenant obtenir un automate déterministe  $(D, \Sigma, \delta_D, d_0, D_F)$  à partir d'un automate non-déterministe  $(N, \Sigma, \delta_N, n_0, N_F)$ . La clef est de dériver  $D$  et  $\delta_D$ .  $\Sigma$  reste le même. L'algorithme utilisé est appelé la *construction de sous-ensembles* (subset construction). L'idée c'est que les états du nouvel automate sont des ensembles d'états de l'ancien automate. Un nouvel état  $q_i = \{n_1, \dots, n_k\}$  contient précisément l'ensemble des anciens états pouvant être atteints en lisant un caractère particulier depuis un état  $q_j$ . L'algorithme utilisé pour construire le nouvel automate est le suivant :

$q_0 \leftarrow \epsilon\text{-fermeture}(n_0)$   
 initialiser  $Q$  avec  $q_0$   
 WorkList  $\leftarrow q_0$

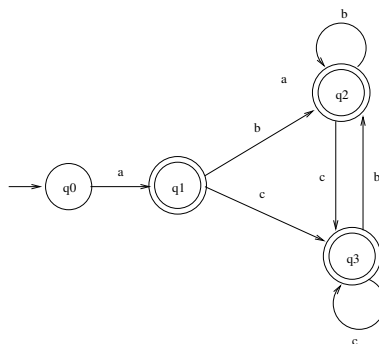
Tant que (WorkList  $\neq \emptyset$ )  
 choisir  $q_i$  dans WorkList  
 pour chaque caractère  $c \in \Sigma$   
 $q \leftarrow \epsilon$ -fermeture( $\Delta(q_i, c)$ )  
 $\delta_D[q_i, c] \leftarrow q$   
 si  $q \notin Q$  alors ajouter  $q$  à WorkList ;  
 ajouter  $q$  à  $Q$  ;

L' $\epsilon$ -fermeture d'un ensemble d'états ajoute à cet ensemble d'état tous les états pouvant être atteint par des  $\epsilon$ -transitions à partir des états de l'ensemble. L'opérateur  $\Delta(q_i, c)$  calcule l'ensemble des états atteignables en lisant  $c$  depuis les états  $n_j^i$  de  $q_i$  : si  $q_i = \{n_1^i, \dots, n_k^i\}$  alors  $\Delta(q_i, c) = \cup_{n_j^i \in q_i} \delta_N(n_j^i, c)$

Appliquons cet algorithme sur l'automate non-déterministe obtenu (on a renommé les états).



1. l' $\epsilon$ -fermeture de  $n_0$  donne l'état  $q_0 = \{n_0\}$ .
2. la première itération de la boucle while calcule :
  - $\Delta(q_0, a) = \{n_1\}$  ;  $\epsilon$ -fermeture( $\{n_1\}$ ) =  $\{n_1, n_2, n_3, n_4, n_6, n_9\} = q_1$
  - $\Delta(q_0, b) = \emptyset$
  - $\Delta(q_0, c) = \emptyset$
3. la deuxième itération de la boucle while calcule :
  - $\Delta(q_1, a) = \emptyset$
  - $\Delta(q_1, b) = \{n_5\}$  ;  $\epsilon$ -fermeture( $\{n_5\}$ ) =  $\{n_5, n_8, n_9, n_3, n_4, n_6\} = q_2$
  - $\Delta(q_1, c) = \{n_7\}$  ;  $\epsilon$ -fermeture( $\{n_7\}$ ) =  $\{n_7, n_8, n_9, n_3, n_4, n_6\} = q_3$
4. les itérations suivantes retomberont sur les mêmes états  $q_2$  et  $q_3$ , l'algorithme s'arrête en 4 itérations et produit l'automate suivant :



On peut montrer que cette méthode pour obtenir des automates déterministes reconnaissant une expression régulière produit des automates avec beaucoup d'états ( $Q$  peut avoir  $2^{|N|}$  états) mais n'augmente pas le nombre de transitions nécessaires pour reconnaître un mot.

**remarque : algorithme de point fixe** L'algorithme que nous venons de voir est un bon exemple d'algorithmes de point fixe qui sont très largement utilisés en informatique. La caractéristique de tels algorithmes est qu'ils appliquent une fonction monotone ( $f(x) \geq x$ ) à une collection d'ensembles choisis dans un domaine ayant une certaine structure. Ces algorithmes terminent lorsque les itérations suivantes de la fonction ne modifient plus le résultat, on parle de point fixe atteint.

La terminaison de tels algorithmes dépend des propriétés du domaine choisi. Dans notre exemple, on sait que chaque  $q_i \in Q$  est aussi dans  $2^N$ , donc ils sont en nombre fini. Le corps de la boucle while est monotone car l'ensemble  $Q$  ne peut que grossir. Donc forcément, au bout d'un moment,  $Q$  s'arrêtera de grossir. Lorsque  $Q$  ne grossit plus, l'algorithme s'arrête en  $|Q|$  itérations au maximum, donc l'algorithme présenté s'arrête bien.

On calcule l' $\epsilon$ -fermeture par un autre algorithme point fixe.

### 2.2.3 Minimisation

Le grand nombre d'états de l'automate résultant de la déterminisation est un problème pour l'implémentation. La minimisation permet de regrouper les états équivalents, c'est à dire qui ont le même comportement sur les mêmes entrées. L'algorithme de minimisation construit une partition de l'ensemble  $D$  des états de l'automate déterministe, chaque état étant dans le même sous-ensemble a le même comportement sur les mêmes entrées. L'algorithme est le suivant :

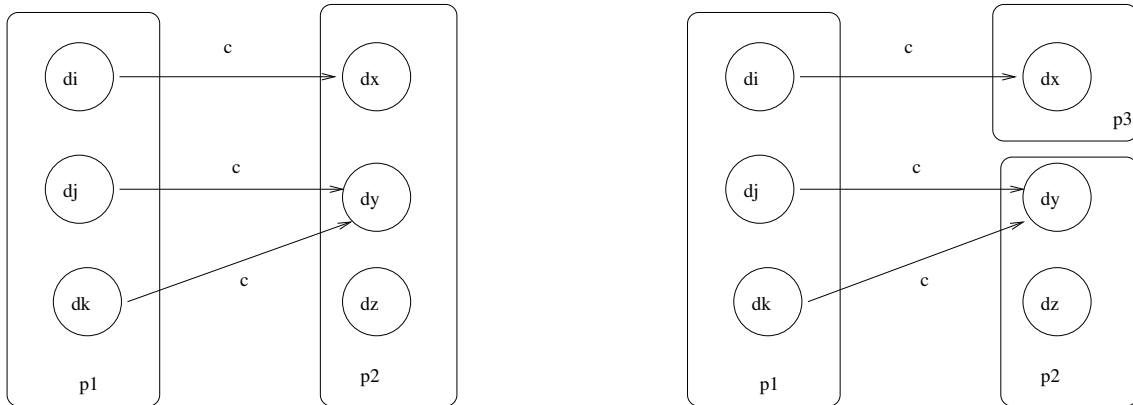
```

P ← {D_F, D - D_F}
tant que (P change)
  T ← ∅
  pour chaque ensemble p ∈ P
    T ← T ∪ Partition(p)
  P ← T
  
```

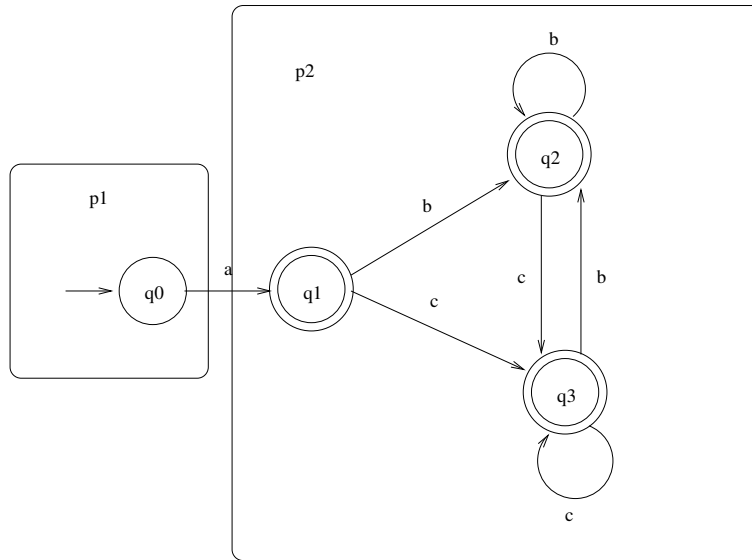
```

Partition(p)
  pour chaque c ∈ Σ
    si c sépare p en {p_1, ..., p_k}
      alors Return({p_1, ..., p_k})
  Return(p)
  
```

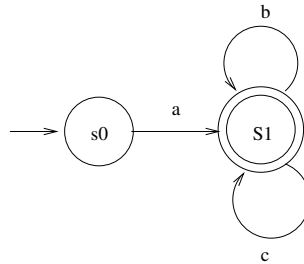
La procédure Partition découpe l'ensemble  $p$  en mettant ensemble les états  $p_i$  de  $p$  qui arrivent dans le même ensemble. Par exemple, le schéma ci-dessous présente deux cas de figure. Sur la gauche, on a un automate dans lequel le caractère  $c$  ne sépare pas la partition  $p_1$  car toutes les transitions étiquetées par  $c$  arrivent sur des états qui sont dans la même partition ( $d_x, d_y, d_z$  sont dans  $p_2$ ). Sur la droite,  $c$  sépare  $p_1$  en  $p'_1 = \{\{d_i\}, \{d_j, d_k\}\}$  car les transitions étiquetées par  $c$  partant de  $d_i$  n'arrivent pas dans la même partition que celles partant de  $d_j$  et  $d_k$ .



Appliquons cet algorithme sur l'exemple, l'algorithme démarre avec la partie élémentaire séparant les états finaux des non finaux :



L'algorithme est incapable de séparer p2, il termine donc immédiatement et on obtient alors l'automate recherché :



On peut aussi construire l'expression régulière qui correspond à un automate donné (c'est la construction de Kleene), c'est ce qui prouve l'équivalence entre langages reconnus par des automates et expressions régulières. Cela est étudié dans le cours de Marianne Delorme.

## 2.3 Implémentation de l'analyseur lexical

Il s'agit maintenant de produire une implémentation de l'automate généré lors de la production précédente. Il y a essentiellement deux types d'implémentation, les analyseurs basés sur des tables et les analyseurs codés directement .

### 2.3.1 Analyseur lexical à base de table

L'analyseur utilise un programme squelette commun à tous les analyseurs lexicaux et une table dans laquelle est stockée la fonction de transition. Pour notre exemple cela donne :

```

char ← prochain caractère
state ← s0
tant que (char ≠ EOF)
    state ← δ(state, char)
    char ← prochain caractère
si (state ∈ SF)
    alors accepter
    sinon rejeter

```

δ	a	b	c	autres
s <sub>0</sub>	s <sub>1</sub>	—	—	—
s <sub>1</sub>	—	s <sub>1</sub>	s <sub>1</sub>	—
—	—	—	—	—

Vous remarquez que la table est très creuse (beaucoup de —). Elle se prêtera donc à des techniques de compression ad-hoc.

### 2.3.2 Analyseur lexical codé directement

Le type d'analyseur précédent passe beaucoup de temps à tester et manipuler la variable d'état, on peut éviter cela en codant directement l'état dans l'état du programme :

```
s0 : char ← prochain caractère
      si (char = ' a' )
          alors goto s1
          sinon goto se

s1 : char ← prochain caractère
      si (char = ' b' | char = ' c' )
          alors goto s1
          sinon si (char = ' EOF' )
              alors accepter
              sinon goto se

se : rejeter
```

Vous constatez que l'état de l'automate n'est plus codé comme une donnée du programme, mais est codé dans la position du pointeur de programme, ou en d'autres termes dans l'état de la machine de Von Neuman sur laquelle s'exécute ce programme.

Ce type d'implémentation devient rapidement difficile à gérer manuellement (le cerveau n'aime pas le "code spaghetti" à base de goto, voir à ce sujet le fameux article "Gotos considered harmful"), mais il peut être généré automatiquement et sera en général plus rapide que l'analyseur à base de table.

### 2.4 Pour aller plus loin (en arrière)

Un analyseur lexical peut faire un peu plus que reconnaître une phrase du langage, il peut préparer le terrain pour l'analyse syntaxique. En particulier, il peut reconnaître les *mots clefs* du langage. Pour cela il y a deux possibilités, soit on introduit une suite explicite d'états reconnaissant chaque mot clef, soit on reconnaît les identificateurs et on les comparera à une table des symboles contenant les mots clés du langage.

En général, l'analyseur lexical va générer une séquence de *mots élémentaires* ou *token* (séparant identificateurs, valeurs, mots clefs, structure de contrôle, etc.) qui sera utilisée pour l'analyse syntaxique.

On peut choisir de réaliser des actions dans chaque état de l'analyseur (pas seulement lorsque l'on a fini de lire la chaîne). Par exemple si on essaie de reconnaître un entier, on peut calculer *au vol* sa valeur pour la produire dès que l'entier est reconnu.

Le développement des techniques présentées ici a entraîné des définitions de langage cohérentes avec les contraintes des expressions régulières. Mais les langages anciens ont certaines caractéristiques qui les rendent délicats à analyser. Par exemple, en Fortran 77, les espaces ne sont pas significatifs, donc `unentier`, `un entier`, `unentier` représentent la même chaîne. Les identificateurs sont limités à 6 caractères, et le compilateur doit utiliser cela pour identifier les structures du programme ; les mots clés ne sont pas réservés. Par exemple `FORMAT(4H)=(3)`

est l'instruction `FORMAT` car `4H` est un préfixe qui désigne `=(3` comme une chaîne de caractère. Alors que `FORMAT(4)=(3)` est l'assignation à l'élément numéro 4 du tableau `FORMAT`.

Ces difficultés ont amené les développeurs à produire des analyseurs lexicaux à deux passes pour Fortran.

### 3 Analyse syntaxique

Dans ce cours nous allons étudier comment produire des analyseurs syntaxique. Pour des raisons d'impérialisme culturel anglo-saxon, on utilisera souvent le terme *parseur* pour analyseur syntaxique et *parser* pour "analyser la syntaxe". C'est vilain, mais c'est plus court.

Plan : on va surtout s'intéresser aux langages définis par des *grammaires hors contexte*. Il existe de nombreuses méthodes pour parser de tels langage, nous verrons une méthode récursive descendante (*top down recursive descent parsing*) qui est la méthode la plus pratique lorsqu'on code le parser à la main (ce que plus personne ne fait sans doute). Les parseurs de ce type seront aussi appelés *LL(1)*. Nous verrons ensuite une technique utilisés par les générateurs automatiques de parseurs tels *yacc*. Cette technique sera dite ascendante (*bottom-up*), ou *LR(1)*, ou *shift reduce parsing* selon votre fantaisie.

À chaque technique est associée un ensemble de grammaire que cette technique peut parser, et on peut montrer par exemple que  $LL(1) \subset LR(1)$ , et d'autres inclusions qui seront discutées à la fin de ce cours.

#### 3.1 Grammaires hors contexte

Pour décrire la syntaxe d'un langage de programmation, on utilise une *grammaire*. Une grammaire est un ensemble de règles décrivant comment former des phrases.

**Définition 3** Une grammaire hors contexte (*context free grammar, CFG*) est un quadruplet  $G = (T, NT, S, P)$  où :

- $T$  est l'ensemble des symboles terminaux du langage. Les symboles terminaux correspondent aux tokens découverts par l'analyseur lexical. On représentera les terminaux soulignés pour les identifier.
- $NT$  est l'ensemble des symboles non-terminaux du langage. Ces symboles n'apparaissent pas dans le langage mais dans les règles de la grammaire définissant le langage, ils permettent d'exprimer la structure des règles grammaticales.
- $S \in NT$  est appelé l'élément de départ de  $G$  (ou axiome de  $G$ ). Le langage que  $G$  décrit (noté  $L(G)$ ) correspond à l'ensemble des phrases qui peuvent être dérivées (ou construites) à partir de  $S$  par les règles de la grammaire.
- $P$  est un ensemble de production (ou règles de réécriture) de la forme  $N \rightarrow \alpha_1\alpha_2 \dots \alpha_n$  avec  $\alpha_i \in T \cup NT$ . C'est à dire que chaque élément de  $P$  associe un non terminal à une suite de terminaux et non terminaux. Le fait que les parties gauches des règles ne contiennent qu'un seul non terminal donne la propriété "hors contexte" à la grammaire.

□

Exemple : une liste d'élément

$$T = \{\underline{elem}\}, NT = \{List\}, S = List, P = \left\{ \begin{array}{l} List \rightarrow \underline{elem} List \\ List \rightarrow \underline{elem} \end{array} \right\}$$

On peut facilement voir qu'une suite finie d'un nombre quelconque de elem correspond à une phrase du langage décrit par cette grammaire. Par exemple, elem elem correspond à l'application de la première règle puis de la deuxième en partant de  $S$  :

$$List \rightarrow \underline{elem} List \rightarrow \underline{elem} \underline{elem}$$

Exemple : des parenthèses et crochets correctement équilibrés en alternance, on pourra utiliser les règles suivants :

$$P = \left\{ \begin{array}{ll} Paren \rightarrow ( Croch ) & Croch \rightarrow [ Paren ] \\ \quad \quad \quad | \quad ( \quad ) & \quad \quad \quad | \quad [ \quad ] \end{array} \right\}$$

On peut éventuellement rajouter un non terminal *Start* pour commencer indifféremment par des crochets ou par des parenthèses :

$$\begin{array}{l} Start \rightarrow Parent \\ \quad \quad \quad | \quad Croch \end{array}$$

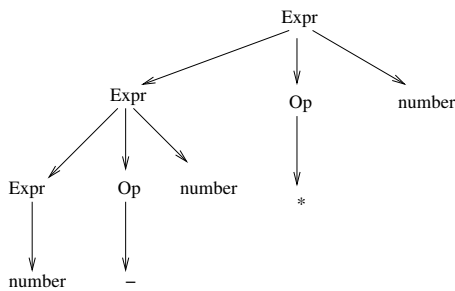
Considérons la grammaire suivante :

1.  $Expr \rightarrow Expr Op \underline{nombre}$
2.  $\quad \rightarrow \underline{nombre}$
3.  $Op \rightarrow \pm$
4.  $\quad \rightarrow \mp$
5.  $\quad \rightarrow \times$
6.  $\quad \rightarrow \div$

En appliquant la séquence de règles : 1, 5, 1, 4, 2 on arrive à dériver la phrase : *nombre – nombre × nombre* :

Règle	Phrase
	<i>Expr</i>
1	<i>Expr Op nombre</i>
5	<i>Expr × nombre</i>
1	<i>Expr Op nombre × nombre</i>
4	<i>Expr – nombre × nombre</i>
2	<i>nombre – nombre × nombre</i>

On peut représenter graphiquement cette dérivation par un arbre que l'on nomme *arbre de dérivation* (ou arbre syntaxique, arbre de syntaxe, parse tree).



Lors de cette dérivation, nous avons toujours décidé d'utiliser une règle dérivant du non terminal le plus à droite de la phrase (dérivation la plus à droite, *rightmost derivation*). On aurait pu arriver à la même phrase en choisissant systématiquement le non-terminal le plus à gauche par exemple (dérivation la plus à gauche), cela donne l'ordre d'application des règles : 1, 1, 2, 4, 5

Règle	Phrase
	<i>Expr</i>
1	<i>Expr Op nombre</i>
1	<i>Expr Op nombre Op nombre</i>
2	<i>nombre Op nombre Op nombre</i>
4	<i>nombre – nombre Op nombre</i>
5	<i>nombre – nombre × nombre</i>

Si l'on dessine l'arbre de dérivation correspondant on s'aperçoit que c'est le *même* arbre que celui dessiné précédemment. En général, ce n'est pas toujours le cas. Cette propriété est appelée la non-ambiguïté. Une grammaire est ambiguë s'il existe une phrase de  $L(G)$  qui possède plusieurs arbres de dérivation.

L'exemple classique de grammaire ambiguë est le if-then-else

- 1  $|$  *Instruction*  $\rightarrow$  *if Expr then Instruction else Instruction*
- 2  $|$  *if Expr then Instruction*
- 3  $|$  *Assigination*
- 4  $|$  *AutreInstructions...*

Avec cette grammaire, le code

*if Expr<sub>1</sub> then if Expr<sub>2</sub> then Ass<sub>1</sub> else Ass<sub>2</sub>*

peut être compris de deux manières :

$\begin{array}{l} \text{if } Expr_1 \\ \quad \text{then if } Expr_2 \\ \quad \quad \text{then } Ass_1 \\ \quad \quad \text{else } Ass_2 \end{array}$	$\begin{array}{l} \text{if } Expr_1 \\ \quad \text{then if } Expr_2 \\ \quad \quad \text{then } Ass_1 \\ \quad \quad \text{else } Ass_2 \end{array}$
--	--

Bien sûr, les sens des deux interprétations sont différents. Pour enlever cette ambiguïté, on doit faire un choix pour la résoudre (par exemple que chaque *else* se rapporte au *if* "non fermé" le plus proche dans l'imbrication), et on doit modifier la grammaire pour faire apparaître cette règle de priorité, par exemple de la façon suivante :

1	<i>Instruction</i>	→	<i>AvecElse</i>
2			<i>DernierElse</i>
3	<i>AvecElse</i>	→	<u>if</u> <i>Expr</i> <u>then</u> <i>AvecElse</i> <u>else</u> <i>AvecElse</i>
4			<i>Assignment</i>
5			<i>AutreInstructions...</i>
6	<i>DernierElse</i>	→	<u>if</u> <i>Expr</i> <u>then</u> <i>Instruction</i>
7			<u>if</u> <i>Expr</i> <u>then</u> <i>AvecElse</i> <u>else</u> <i>DernierElse</i>
8			<i>AutreInstructions...</i>

On voit donc une relation entre la structure grammaticale et le sens d'une phrase. Il existe d'autres situations pour lesquelles la structure de la grammaire va influencer sur la manière avec laquelle on *comprend la phrase*. Par exemple, pour la priorité des opérateurs, la grammaire donnée précédemment pour les expressions arithmétiques simples ne permet pas d'encoder dans l'arbre de dérivation l'ordre correct d'évaluation de l'expression : si l'on parcourt l'arbre d'un manière naturelle par exemple par un parcours en profondeur, on évaluera l'expression  $(\text{nombre} - \text{nombre}) \times \text{nombre}$  alors que l'on voudrait évaluer  $\text{nombre} - (\text{nombre} \times \text{nombre})$ .

Pour permettre que l'arbre de dérivation reflète la précedence des opérateurs, il faut restructurer la grammaire, par exemple de la façon suivante :

1.	<i>Expr</i>	→	<i>Expr</i> <u>+</u> <i>Term</i>
2.		→	<i>Expr</i> <u>-</u> <i>Term</i>
3.		→	<i>Term</i>
4.	<i>Term</i>	→	<i>Term</i> <u>×</u> <i>number</i>
5.		→	<i>Term</i> <u>÷</u> <i>number</i>
6.		→	<i>number</i>

On fera de même pour introduire les précedences supérieures pour les parenthèses, les indices de tableaux, les conversions de type (*type-cast*) etc. Dans la suite on utilisera la grammaire ci-dessous (grammaire d'expressions classiques)

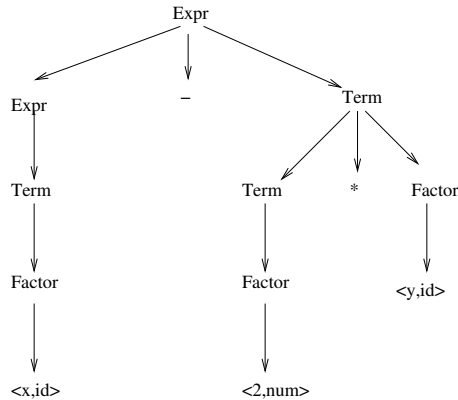
1.	<i>Expr</i>	→	<i>Expr</i> <u>+</u> <i>Term</i>
2.		→	<i>Expr</i> <u>-</u> <i>Term</i>
3.		→	<i>Term</i>
4.	<i>Term</i>	→	<i>Term</i> <u>×</u> <i>Factor</i>
5.		→	<i>Term</i> <u>÷</u> <i>Factor</i>
6.		→	<i>Factor</i>
7.	<i>Factor</i>	→	( <i>Expr</i> )
8.		→	<i>number</i>
9.		→	<i>identifiant</i>

Jusqu'à présent nous avons "deviné" les règles à utiliser pour dériver une phrase. Le travail du parser est de construire l'arbre de dérivation à partir d'une phrase du langage composée donc uniquement de terminaux. En fait, le parser prend en entrée une phrase "allégée" par l'analyse syntaxique c'est à dire une séquence de mots (token) avec leur catégorie syntaxique attachée. Par exemple si le code source est  $x - 2 \times y$ , l'entrée du parser sera quelque chose comme :

$\langle \text{identificateur}, "x" \rangle - \langle \text{nombre}, "2" \rangle \times \langle \text{identificateur}, "y" \rangle$



On voudrait produire l'arbre suivant grâce à la grammaire des expressions arithmétiques classiques.



Les parseurs descendants (*top down parsers*) vont partir de l'axiome et à chaque étape choisir d'appliquer une règle pour l'un des non terminaux présents sur la *frontière* de l'arbre. Les parseurs ascendants (*bottom up parsers*) vont partir des feuilles et tenter de reconstruire l'arbre à partir du bas.

Les grammaires hors contexte décrivent plus de langages que les expressions régulières (ce lui des parenthèses équilibrées ne peut être exprimé en expression régulière, les priorités entre opérateurs peuvent être exprimées dans la structure des grammaires hors contexte). La classe des grammaires *régulières* décrit exactement les mêmes langages. Une grammaire est dite *régulière* (ou grammaire *linéaire à gauche*) si toutes les productions sont de la forme : soit  $A \rightarrow a$  soit  $A \rightarrow aB$  avec  $A, B \in NT$  et  $a \in T$ . Les langages réguliers sont un sous ensemble strict de l'ensemble des langages reconnus par les grammaires hors contexte.

On peut légitimement se demander pourquoi on n'utilise pas uniquement le parsing pour analyser un programme source. A-t-on besoin de l'analyse lexicale ? La raison est que les analyseurs basés sur des automates finis sont très efficaces, le temps d'analyse est proportionnel à la longueur de la chaîne d'entrée et les constantes multiplicatives sont très petites (un analyseur syntaxique serait beaucoup plus long). Mais surtout, en reconnaissant les structures lexicales de base, les analyseurs lexicaux réduisent fortement la complexité des grammaires nécessaires à spécifier la syntaxe du langage (suppression des commentaires, détection des identificateurs, des valeurs etc.). Enfin, c'est naturel pour un programmeur d'exprimer cette structure à deux étages (on dit parfois microsyntaxe et macrosyntaxe) puisque c'est celle des langues naturelles à alphabet.

En résumé de cette partie, il existe plusieurs grammaires qui reconnaissent un même langage, et lorsqu'on a des propriétés implicites du langage (qui sont en fait des propriétés de l'arbre qu'on veut associer à une phrase) on préférera une grammaire à une autre.

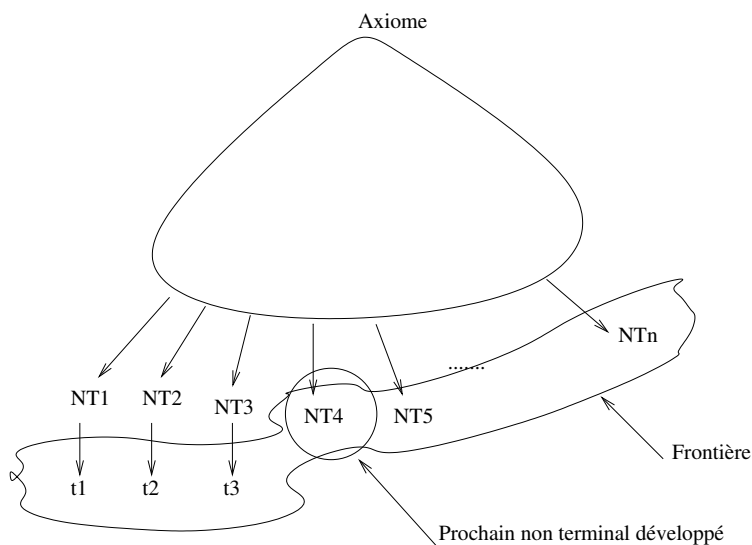
### 3.2 Analyse syntaxique descendante

Si vous aviez à coder un parseur pour notre grammaire des expressions naturelles dans votre Caml chéri, sans avoir l'idée qu'il existe des cours de compilation, vous aimeriez sans doute refléter la structure récursive de la grammaire dans celle du code. Typiquement, vous aimeriez implémenter une règle comme  $Expr \rightarrow Expr \pm Term \rightarrow \dots$  par une fonction `ReconnaisExpr` qui ferait appel récursivement à elle-même, et à des fonctions `ReconnaisTerm`, etc. Bref, on écrirait une fonction par non-terminal (et une fonction par terminal qui serait triviale). Dans ce chapitre on va voir que cela marche parfois (rarement), que lorsque cela ne marche pas (récursion infinie) il est parfois possible de bricoler la grammaire pour que cela marche.

Le principe de l'analyse syntaxique descendante est donc le suivant : on commence avec l'axiome de la grammaire (appel de la fonction `ReconnaisAxiom`) et on développe systématiquement l'arbre de dérivation jusqu'aux feuilles (terminaux).

Comme on aimerait manger le texte à parser de gauche à droite, on choisit systématiquement de dériver le non-terminal le plus à gauche sur la frontière de l'arbre. Quand on arrive aux terminaux, on vérifie qu'ils correspondent aux tokens qui sont dans la phrase ; si c'est OK, on conti-

nue, sinon, il faut revenir en arrière (*backtracker*) et choisir une autre règle dans le processus de dérivation. L'arbre de dérivation au cours de l'algorithme ressemble donc à cela :



On va voir que ce qui rend ce processus viable est qu'il existe un large sous ensemble de grammaires hors contexte pour lesquelles on n'aura jamais besoin de backtracker. On utilise un pointeur sur la chaîne à lire qui ne peut qu'avancer (on n'a pas accès à n'importe quel endroit de la chaîne n'importe quand).

### 3.2.1 Exemple d'analyse descendante

Considérons que l'on ait à parser l'expression  $x - 2 \times y$  que l'on veut parser avec la grammaire des expressions arithmétiques classiques. On va utiliser les notation précédentes pour la succession de règles permettant la dérivation en ajoutant un signe  $\rightarrow$  dans la colonne des numéros de règle pour indiquer que l'on avance le pointeur de lecture. La flèche verticale indiquera la position du pointeur de lecture. Si l'on applique bêtement la méthode expliquée ci-dessus, on va pouvoir tomber sur ce type de dérivation :

Règle	Phrase	Entree
	<i>Expr</i>	$\uparrow x - 2 \times y$
1	<i>Expr</i> + <i>Term</i>	$\uparrow x - 2 \times y$
1	<i>Expr</i> + <i>Term</i> + <i>Term</i>	$\uparrow x - 2 \times y$
1	<i>Expr</i> + <i>Term</i> ...	$\uparrow x - 2 \times y$
1	...	$\uparrow x - 2 \times y$

En fait la grammaire telle qu'elle est définie pose un problème intrinsèque car elle est récursive à gauche. Ignorons pour l'instant ce problème et supposons que le parser ait effectué la dérivation suivante :

Règle	Phrase	Entree
	<i>Expr</i>	$\uparrow x - 2 \times y$
3	<i>Term</i>	$\uparrow x - 2 \times y$
6	<i>Factor</i>	$\uparrow x - 2 \times y$
9	<i>identifiant</i>	$\uparrow x - 2 \times y$
$\rightarrow$	<i>identifiant</i>	$x \uparrow - 2 \times y$

Ici, on ne boucle pas mais on n'a pas reconnu la phrase entière, on en a reconnu une partie, mais on n'a plus de non terminal à remplacer, on ne sait pas si l'on a fait un mauvais choix à un moment ou si la phrase n'est pas valide. Comme il reste un caractère à droite du  $\rightarrow$ , on s'est sans doute planté, donc il faut backtracker et, en supposant que l'on puisse essayer toutes les possibilités, on arrivera finalement à la dérivation suivante :

Règle	Phrase	Entree
	$Expr$	$\uparrow x - 2 \times y$
2	$Expr - Term$	$\uparrow x - 2 \times y$
3	$Term - Term$	$\uparrow x - 2 \times y$
6	$Factor - Term$	$\uparrow x - 2 \times y$
9	$identif\grave{e}r - Term$	$\uparrow x - 2 \times y$
$\rightarrow$	$identif\grave{e}r - Term$	$x \uparrow - 2 \times y$
$\rightarrow$	$identif\grave{e}r - Term$	$x - \uparrow 2 \times y$
4	$identif\grave{e}r - Term \times Factor$	$x - \uparrow 2 \times y$
6	$identif\grave{e}r - Factor \times Factor$	$x - \uparrow 2 \times y$
8	$identif\grave{e}r - number \times Factor$	$x - \uparrow 2 \times y$
$\rightarrow$	$identif\grave{e}r - number \times Factor$	$x - 2 \uparrow \times y$
$\rightarrow$	$identif\grave{e}r - number \times Factor$	$x - 2 \times \uparrow y$
9	$identif\grave{e}r - number \times identif\grave{e}r$	$x - 2 \times \uparrow y$
$\rightarrow$	$identif\grave{e}r - number \times identif\grave{e}r$	$x - 2 \times y \uparrow$

### 3.2.2 Élimination de la récursion à gauche

La récursivité à gauche pose un problème pour ce type de parser car le backtrack ne peut arriver que lorsqu'on arrive sur un non terminal en première position dans la *phrase partielle* en cours de reconnaissance. On peut transformer mécaniquement des grammaires pour enlever toute récursivité à gauche en autorisant les règles du type  $A \rightarrow \epsilon$ .

$$\begin{array}{l}
 A \rightarrow A \underline{\alpha} \\
 \rightarrow \underline{\beta}
 \end{array}
 \qquad
 \begin{array}{l}
 A \rightarrow \underline{\beta} B \\
 B \rightarrow \underline{\alpha} B \\
 \rightarrow \epsilon
 \end{array}$$

Lorsque la récursivité est indirecte, on *déroule* les règles jusqu'à ce que l'on rencontre une récursivité directe et on utilise le même processus :

$$\begin{array}{l}
 A \rightarrow B \underline{\alpha} \\
 B \rightarrow A \underline{\beta} \\
 \rightarrow \underline{\gamma}
 \end{array}
 \qquad
 \begin{array}{l}
 A \rightarrow A \underline{\beta} \underline{\alpha} \\
 \rightarrow \underline{\gamma} \underline{\alpha} \\
 B \rightarrow A \underline{\beta} \\
 \rightarrow \underline{\gamma}
 \end{array}
 \qquad
 \begin{array}{l}
 A \rightarrow \underline{\gamma} \underline{\alpha} C \\
 C \rightarrow \underline{\beta} \underline{\alpha} C \\
 \rightarrow \epsilon \\
 B \rightarrow A \underline{\beta} \\
 \rightarrow \underline{\gamma}
 \end{array}$$

Du fait de la nature finie de la grammaire, on sait que le processus de déroulage va s'arrêter. La grammaire des expressions arithmétiques classique peut être réécrite en variante récursive à droite :

1.  $Expr \rightarrow Term Expr'$
2.  $Expr' \rightarrow \underline{+} Term Expr'$
3.  $\rightarrow \underline{-} Term Expr'$
4.  $\rightarrow \epsilon$
5.  $Term \rightarrow Factor Term'$
6.  $Term' \rightarrow \underline{\times} Factor Term'$
7.  $\rightarrow \underline{\div} Factor Term'$
8.  $\rightarrow \epsilon$
9.  $Factor \rightarrow ( Expr )$
10.  $\rightarrow number$
11.  $\rightarrow identif\grave{e}r$

Vous me direz que c'est moins naturel, comme description de la grammaire, et vous aurez raison. Le problème est que la grammaire naturelle est récursive à gauche parce que cela décrit le parenthésage implicite à gauche.

### 3.2.3 Élimination du backtrack

On va montrer comment on peut, pour des grammaires récursives à droite, systématiquement éliminer le backtrack. En fait, le backtrack arrive lorsque l'algorithme choisit une mauvaise règle à appliquer. Si l'algorithme choisissait systématiquement la bonne règle, il n'y aurait jamais besoin de backtracker.

L'idée est que lorsque le parser choisit une règle pour remplacer le non terminal le plus à gauche sur la frontière, il va être capable de connaître l'ensemble des terminaux qui arriveront finalement à cette place. En regardant le premier token non accepté sur la phrase d'entrée, le parser va alors pouvoir choisir la règle qui convient. On peut montrer que pour les grammaires récursives à droite, la lecture d'un seul token en avant permet de décider quelle règle appliquer. De là vient le terme  $LL(1)$  : *Left-to-right scanning, Leftmost derivation, use 1 symbol lookahead*. Remarquez que le premier  $L$  correspond au fait que nous lisons de gauche à droite, et on l'aura pour toutes nos grammaires.

Voyons comment cela marche en reconnaissant l'expression  $x - 2 \times y$  avec la version récursive à droite de la grammaire d'expressions arithmétiques classiques :

Règle	Phrase	Entree
	$Expr$	$\uparrow x - 2 \times y$
1	$Term Expr'$	$\uparrow x - 2 \times y$
5	$Factor Term' Expr'$	$\uparrow x - 2 \times y$
11	$id Term' Expr'$	$\uparrow x - 2 \times y$
$\rightarrow$	$id Term' Expr'$	$x \uparrow - 2 \times y$
8	$id Expr'$	$x \uparrow - 2 \times y$
3	$id - Term Expr'$	$x \uparrow - 2 \times y$
$\rightarrow$	$id - Term Expr'$	$x - \uparrow 2 \times y$
5	$id - Factor Term' Expr'$	$x - \uparrow 2 \times y$
10	$id - num Term' Expr'$	$x - \uparrow 2 \times y$
$\rightarrow$	$id - num Term' Expr'$	$x - 2 \uparrow \times y$
6	$id - num \times Factor Term' Expr'$	$x - 2 \uparrow \times y$
$\rightarrow$	$id - num \times Factor Term' Expr'$	$x - 2 \times \uparrow y$
11	$id - num \times id Term' Expr'$	$x - 2 \times \uparrow y$
$\rightarrow$	$id - num \times id Term' Expr'$	$x - 2 \times y \uparrow$
8	$id - num \times id Expr'$	$x - 2 \times y \uparrow$
4	$id - num \times id$	$x - 2 \times y \uparrow$

Les deux premières dérivations sont imposées. Nous voyons que la troisième dérivation (application de la règle 11) est imposée par le fait que le token arrivant n'est ni une parenthèse ni un nombre. De même on est obligé d'appliquer la règle 8 ( $Term' \rightarrow \epsilon$ ) car sinon on devrait trouver soit un  $\times$  soit un  $\div$ . On continue jusqu'à reconnaissance de la phrase entière.

On va formaliser cela en introduisant les ensembles *First* et *Follow*.

**Définition 4** Soit  $First(\alpha)$  l'ensemble des symboles terminaux qui peuvent apparaître comme premier symbole dans une phrase dérivée du symbole  $\alpha$ . Si  $\alpha$  peut dériver  $\epsilon$ , alors  $\epsilon \in First(\alpha)$ .  $\square$

*First* est défini pour les symboles terminaux et non terminaux. Considérons les règles pour  $Expr'$  :

$$\begin{array}{l} 2. \left| \begin{array}{l} Expr' \rightarrow \pm Term Expr' \\ \phantom{Expr'} \rightarrow \_ Term Expr' \\ \phantom{Expr'} \rightarrow \epsilon \end{array} \right. \end{array}$$

Pour les deux premières, il n'y a pas d'ambiguïtés, par contre, pour l' $\epsilon$  transition, le seul symbole dérivé est  $\epsilon$ , on met alors  $\epsilon$  dans l'ensemble *First* on a donc :  $First(Expr') = \{+, -, \epsilon\}$ .

Si un ensemble  $First(A)$  contient  $\epsilon$ , on ne sait pas quel non terminal sera rencontré lorsque ce  $A$  aura été développé. Dans ce cas, le parser doit connaître ce qui peut apparaître immédiatement à la droite de la chaîne vide  $\epsilon$  produite, ici, immédiatement à la droite de  $Expr'$ .

**Définition 5** Étant donné un symbole non terminal  $A$ , on définit  $Follow(A)$  comme l'ensemble des symboles terminaux qui peuvent apparaître immédiatement après un  $A$  dans une phrase valide du langage.

□

Dans notre grammaire, on voit que rien ne peut apparaître après  $Expr'$  à part une parenthèse fermante :  $Follow(Expr') = \{eof, \})\}$ . Voici les algorithmes pour calculer les ensembles  $First$  et  $Follow$  :

```

pour chaque  $\alpha \in T$ 
     $First(\alpha) \leftarrow \alpha$ 
pour chaque  $A \in NT$ 
     $First(A) \leftarrow \emptyset$ 
tant que (les ensembles  $First$  changent)
    Pour chaque  $p \in P$  ou  $p$  à la forme  $A \rightarrow \beta_1\beta_2 \dots \beta_k$ 
         $First(A) \leftarrow First(A) \cup (First(\beta_1) - \{\epsilon\})$ 
         $i \leftarrow 1$ 
        tant que ( $\epsilon \in First(\beta_i)$  et  $i \leq k - 1$ )
             $First(A) \leftarrow First(A) \cup (First(\beta_{i+1}) - \{\epsilon\})$ 
             $i \leftarrow i + 1$ 
        si  $i = k$  et  $\epsilon \in First(\beta_k)$ 
            alors  $First(A) \leftarrow First(A) \cup \{\epsilon\}$ 

pour chaque  $A \in NT$ 
     $Follow(A) \leftarrow \emptyset$ 
tant que (les ensembles  $Follow$  changent)
    Pour chaque  $p \in P$  ou  $p$  à la forme  $A \rightarrow \beta_1\beta_2 \dots \beta_k$ 
        si  $\beta_k \in NT$  alors  $Follow(\beta_k) \leftarrow Follow(\beta_k) \cup Follow(A)$ 
         $Traqueur \leftarrow Follow(A)$ 
        Pour  $i \leftarrow k$  jusqu'à 2
            si  $\epsilon \in First(\beta_i)$  alors
                si  $\beta_{i-1} \in NT$  alors  $Follow(\beta_{i-1}) \leftarrow Follow(\beta_{i-1}) \cup (First(\beta_i) - \{\epsilon\}) \cup Traqueur$ 
            sinon
                si  $\beta_{i-1} \in NT$  alors  $Follow(\beta_{i-1}) \leftarrow Follow(\beta_{i-1}) \cup First(\beta_i)$ 
                 $Traqueur \leftarrow First(\beta_i)$ 

```

Ces algorithmes sont formulés comme des calculs de point fixe, on peut montrer qu'ils convergent (comme d'hab : tout est fini, et les ensembles grossissent, et on a bien ajouté tout ce qu'on pouvait à chaque itération). Les ensembles  $First$  et  $Follow$  pour les non terminaux de la grammaire d'expressions arithmétiques classiques (variante récursive à droite) sont :

symbole	$First$	$Follow$
$Expr$	$\{(\underline{\quad}, number, identifier)\}$	$\{\}$
$Expr'$	$\{\pm, \underline{\quad}, \epsilon\}$	$\{\}$
$Term$	$\{(\underline{\quad}, number, identifier)\}$	$\{\pm, \underline{\quad}, \})\}$
$Term'$	$\{\times, \underline{\quad}, \epsilon\}$	$\{\pm, \underline{\quad}, \})\}$
$Factor$	$\{(\underline{\quad}, number, identifier)\}$	$\{\times, \underline{\quad}, \pm, \underline{\quad}, \})\}$

On peut maintenant formuler la condition à laquelle une grammaire permet d'éviter les back-track dans un parsing descendant : pour tout non-terminal  $A$  qui est en partie de gauche de plusieurs règles :  $A \rightarrow \beta_1 \mid \beta_2 \mid \dots \beta_n$ , on doit avoir la propriété suivante :

$$First^+(\beta_i) \cap First^+(\beta_j) = \emptyset, \quad \forall i, j \quad 1 \leq i < j \leq n$$

en notant  $First^+(\alpha)$  l'ensemble  $First(\alpha)$  si  $\epsilon \notin First(\alpha)$  et  $First(\alpha) \cup Follow(\alpha)$  si  $\epsilon \in First(\alpha)$ . Par extension  $First^+(ABCD)$  désigne  $First^+(A)$ . Cette propriété garantit que le parseur n'hésitera entre réduire par  $\beta_i$  ou par  $\beta_j$ .

Pour que l'on ait cette propriété, étant donnée une grammaire réursive à droite, on va la *factoriser à gauche*, c'est à dire que lorsqu'on aura plusieurs productions pour le même non-terminal qui ont le même préfixe, on va introduire un nouveau non-terminal pour tous les suffixes suivant ce préfixe, ce qui permettra de choisir une et une seule règle lorsqu'on veut démarrer par ce préfixe. Par exemple, si on a les règles :

$$A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \mid \dots \mid \alpha \beta_n \mid \gamma_1 \mid \dots \mid \gamma_k$$

on les remplacera par :

$$\begin{aligned} A &\rightarrow \alpha B \mid \gamma_1 \mid \dots \mid \gamma_k \\ B &\rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n \end{aligned}$$

La factorisation à gauche permet de convertir certaines grammaires nécessitant du backtrack en des grammaires ne nécessitant pas de backtrack (ou grammaires prédictives, ou grammaire LL(1)). Cependant toutes les grammaires hors contexte ne peuvent pas être transformées en grammaires ne nécessitant pas de backtrack. Savoir si une grammaire LL(1) équivalente existe pour une grammaire hors contexte arbitraire est indécidable.

### 3.2.4 Parser descendant récursif

On a maintenant tous les outils pour construire un parser récursif descendant à la main. On va écrire un ensemble de procédures mutuellement récursives, une pour chaque non-terminal dans la grammaire. La procédure pour le non terminal  $A$  doit reconnaître une instance de  $A$  pour cela elle doit reconnaître les différents non-terminaux apparaissant en partie droite d'une production de  $A$ . Par exemple, si  $A$  possède les productions suivantes :

$$A \rightarrow a_1 \alpha_1 \mid \beta_1 \beta_2 \mid \epsilon$$

avec  $a_1 \in T$ ,  $\alpha_1, \beta_1, \beta_2 \in NT$  alors le code de la procédure pour  $A$  aura la forme suivante :

```

trouverA()
  si (tokenCourant =  $a_1$ ) alors
    tokenCourant = prochainToken();
    return (trouver $\alpha_1$ ()) /*  $A \rightarrow a_1 \alpha_1$  choisi */
  sinon si (tokenCourant  $\in$  First+( $\beta_1$ )) alors
    return (trouver $\beta_1$ ()  $\wedge$  trouver $\beta_2$ ()) /*  $A \rightarrow \beta_1 \beta_2$  choisi */
  sinon si (tokenCourant  $\in$  Follow( $A$ ))
    alors return True /*  $A \rightarrow \epsilon$  choisi */
    sinon return False /* erreur */

```

Par exemple, voici le parser récursif descendant pour la grammaire des expressions arithmétiques (variante réursive à droite) :

```

Main()
  /* But → Expr */
  tokenCourant = prochainToken();
  si (Expr() ∧ tokenCourant == eof)
    alors return True
    sinon return False

Expr()
  /* Expr → Term Expr' */
  si (Term() == False)
    alors return False
    sinon return EPrime()

EPrime()
  /* Expr' → + Term Expr' */
  /* Expr' → - Term Expr' */
  si (tokenCourant == +)∨
    (tokenCourant == -) alors
    tokenCourant = prochainToken()
    si (Term() == False)
      alors return False
      sinon return EPrime()
  /* Expr' → ε */
  return True

Term()
  /* Term → Factor Term' */
  si (Factor() == False)
    alors return False
    sinon return TPrime()

EPrime()
  /* Term' → × Factor Term' */
  /* Term' → ÷ Factor Term' */
  si (tokenCourant == ×)∨
    (tokenCourant == ÷) alors
    tokenCourant = prochainToken()
    si (Factor() == False)
      alors return False
      sinon return TPrime()
  /* Term' → ε */
  return True

Factor()
  /* Factor → ( Expr ) */
  si (tokenCourant == () alors
    tokenCourant = prochainToken()
    si (Expr() == False)
      alors return False
      sinon si (tokenCourant ≠))
        alors return False
  sinon
  /* Factor → number */
  /* Factor → identifieur */
  si (tokenCourant() ≠ number)∧
    (tokenCourant() ≠ identifieur)
    alors return False
  tokenCourant = prochainToken()
  return True

```

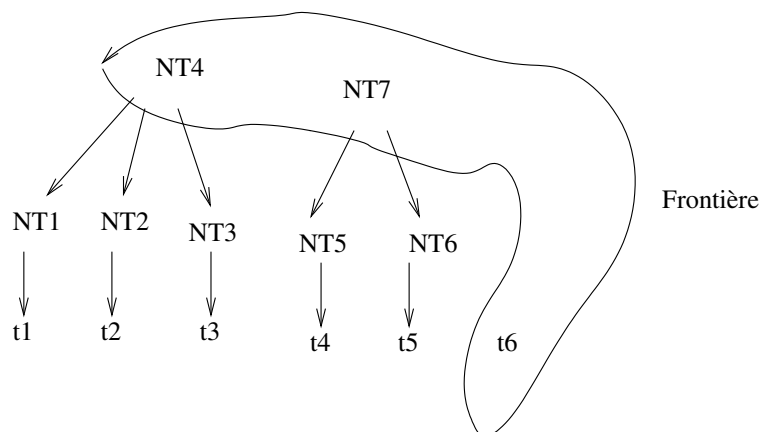
On peut aussi imaginer une implémentation à base de table indiquant, pour chaque non terminal, la règle à appliquer en fonction du token rencontré.

En conclusion, les parseurs *LL(1)* présentent les avantages d'être facilement codés manuellement, d'être compacts et efficaces. Ils permettent également des messages d'erreurs très utiles pour les entrées non conformes, du genre *Pour construire XXX j'attendais YYY et j'ai lu ZZZ*.

Leur gros inconvénient est qu'ils nécessitent un prébricolage de la grammaire (mise sous forme récursive à droite, puis factorisation à gauche) qui, outre que son succès est indécidable, l'éloigne de l'intuition et rend la maintenance (l'extension du langage) difficile. Ceci met aussi un bémol sur la qualité des messages d'erreur : ceux-ci sont précis mais font référence à une grammaire contre-intuitive.

### 3.3 Parsing ascendant

On va voir maintenant l'approche inverse : on construit des feuilles au fur et à mesure qu'on rencontre des tokens, et lorsque l'on peut *réduire* les feuilles en utilisant une règle (i.e. ajouter un parent commun à plusieurs noeuds contigus correspondant à une partie gauche de règle), on le fait. Si la construction est bloquée avant d'arriver à une racine unique étiquetée par l'axiome, le parsing échoue. L'arbre de dérivation est donc construit par le bas et la frontière est maintenant dirigée vers le haut :



Du fait que les réductions de règles ont lieu à partir de la gauche (dans l'ordre de lecture de la phrase), et que l'arbre est construit à l'envers, par les feuilles, l'arbre de dérivation découvert par ce parseur sera un arbre de dérivation la plus à droite. On va considérer les parser  $LR(1)$ . Ces parseurs lisent de gauche à droite et construisent (à l'envers) une dérivation la plus à droite en regardant au plus un symbole sur l'entrée, d'où leur nom : *Left-to-right scan*, *Reverse-rightmost derivation with 1 symbol lookahead*.

### 3.3.1 Survol de l'algorithme

Lorsque, à droite de la frontière, il y a une possibilité de réduction avec la règle  $A \rightarrow \beta$ , on appelle cela une *réduction candidate* (*handle* en anglais).

L'implémentation du parseur va utiliser une pile pour conserver la mémoire de ce qui a été lu et éventuellement réduit. Autrement dit, la frontière sera disponible sur la pile, avec la partie la plus à droite en haut de la pile (pour que le parseur l'ait sous la main). L'itération du parseur consiste alors en chercher une réduction candidates au sommet de pile. S'il y en a une, il la réduit : il dépile les parties droites et empile la partie gauche, ce qui donne sur la pile une nouvelle frontière qui est montée d'un cran vers la racine. Quand cela n'est plus possible, il lit le prochain token et l'empile, puis recommence. L'algorithme générique pour le *shift-reduce* parsing est donc le suivant :

```

push Invalid
tokenCourant ← prochainToken()
tant que (tokenCourant ≠ eof ou la pile ne contient pas exactement Axiom au dessus de Invalid)
  si une réduction candidate  $A \rightarrow \beta$  est au sommet de pile
    alors /* réduire par  $A \rightarrow \beta$  */
      pop  $|\beta|$  symbols
      push A
  sinon si (tokenCourant ≠ eof)
    alors /* shift : décalage d'un sur l'entrée */
      push tokenCourant
      tokenCourant ← prochainToken()
  sinon /* pas de réduction candidate pas d'input */
    error

```

Ici la bonne réduction candidate a été trouvée par un oracle. Une grammaire sera  $LR(1)$  si on peut, dans chaque état du parseur, savoir *uniquement en regardant un token en avant* s'il y a une réduction candidate en haut de la pile (et laquelle), ou s'il faut empiler le token qui arrive. En fait on va définir un automate dont les états sont exactement des ensembles de réductions candidates. L'automate fera des transition en fonction du token qui arrive, et fera aussi des actions sur les transitions. Ces actions seront soit empiler le token qui arrive (*shift*), ou réduire la réduction candidate en haut de la pile (*reduce*).



### 3.3.2 Détection des réductions candidates

On va introduire une nouvelle notation pour les réductions candidates :  $\langle A \rightarrow \beta \bullet \rangle$ . Les lettres grecques représenteront des séquences d'éléments de  $T \cup NT$ . Le point noir ( $\bullet$ , *dot* en anglais) représente le sommet de la pile. La notation  $\langle A \rightarrow \beta \bullet \rangle$  signifie que la pile est telle que l'on peut réduire avec la règle  $A \rightarrow \beta$  à partir du sommet de pile. Dans le même ordre d'idée, la notation  $\langle A \rightarrow \alpha \bullet \gamma \rangle$  va représenter l'état dans lequel "j'ai empilé  $\alpha$ , si jamais j'empile encore  $\gamma$  alors la règle  $A \rightarrow \alpha \gamma$  sera une réduction candidate et je pourrai réduire". On dit alors que l'on a une *réduction potentielle* (elle n'est pas encore candidate puisqu'on a pas lu tout ce dont on aurait besoin pour faire la réduction).

On peut construire un nombre fini de tel *dotted item* (réduction candidate ou potentielle) pour une grammaire donnée. On les appelle aussi *LR-item* ou encore *LR(0)-item*. Les parseurs *LR* sont en fait des automates dont les états sont constitués d'ensembles d'items à boulettes, ou en français des ensembles d'hypothèses sur la phrase en cours de lecture, ou encore des ensembles de réductions potentielles ou candidates. Un état représente en général plusieurs telles hypothèses, et les transitions *shift* de l'automate affinent cet ensemble en lisant des tokens sur l'entrée jusqu'à ce qu'il n'y ait plus qu'une réduction possible, auquel cas on la fait (transition *reduce*).

Le parseur *LR(0)* fait une réduction dès qu'il peut (il ne regarde pas le token qui arrive pour décider s'il faut faire un shift ou une réduction). Lorsqu'il peut hésiter entre un shift et une réduction, on dit qu'il y a un *conflit shift/reduce*. Lorsqu'il peut faire deux réductions différentes, on parle de conflit *reduce/reduce*. Une grammaire sans tels conflits est dite *LR(0)*.

Pour lever certains conflits, le parseur *LR(1)* ne fait les transitions que sous condition que le token suivant le permette.

La grosse difficulté va être de construire deux tables : *Goto*, qui est la table de transition de l'automate, et *Action* qui va indiquer l'action à réaliser à chaque étape. Ces tables contiennent la connaissance "précompilée" des réductions potentielles et candidates pouvant se présenter en fonction de ce qu'on lit et de l'état dans lequel on se trouve. L'algorithme pour le parser *LR(1)* est le suivant :

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

On voit que, au lieu de la phrase vague : "si une réduction candidate  $A \rightarrow \beta$  est au sommet de pile ..." on a une action indiquée par la table Action. Par ailleurs, au lieu d'empiler uniquement la frontière, on empile aussi l'état courant. Autrement dit, chaque étage de la pile porte deux informations : un état et un élément de  $T \cup NT$ . Ainsi, après une réduction  $A \rightarrow \beta$ , on retrouve, après avoir dépilé les tokens correspondant à  $\beta$  que l'on vient de reconnaître, un vieil état empilé jadis, et à partir duquel on va pouvoir choisir l'état suivant sachant à présent qu'on vient de construire  $A$ .

### 3.3.3 Construction des tables LR(1)

Pour construire les tables *Action* et *Goto*, il faut construire un automate permettant de reconnaître les réductions potentielles et candidates. On déduira de cet automate les tables. Cet automate utilise des ensembles d'*item LR(1)* (voir définition ci-après) comme états, l'ensemble des états est appelé *la collection canonique d'ensemble d'item LR(1)* :  $\mathcal{CC} = \{cc_0, cc_1, \dots, cc_n\}$ . Chacun de ces ensembles va correspondre à un état  $s_i$  de l'automate dans l'algorithme ci-dessus.

**Définition 6** Un *item LR(1)* est une paire :  $[A \rightarrow \beta \bullet \gamma, a]$  où  $A \rightarrow \beta \bullet \gamma$  est une réduction potentielle ou candidate,  $\bullet$  indique ou est le sommet de pile et  $a \in T$  est un terminal (on rajoute toujours le terminal EOF pour signifier la fin du fichier).  $\square$

Ces *item LR(1)* décrivent les configurations dans lesquelles peut se trouver le parser ; ils décrivent les réductions potentielles compatibles avec ce qui a été lu, et  $a$  est le prochain token à lire, une fois que l'on aura réduit par cette réduction potentielle. Reprenons l'exemple de la grammaire de liste :

1.  $But \rightarrow List$
2.  $List \rightarrow \underline{elem} List$
3.  $\quad \quad | \underline{elem}$

elle produit les *items LR(1)* suivants :

$$\begin{array}{ll}
 [But \rightarrow \bullet List, eof] & [List \rightarrow \bullet \underline{elem} List, eof] \\
 [But \rightarrow List \bullet, eof] & [List \rightarrow \underline{elem} \bullet List, eof] \\
 [List \rightarrow \bullet \underline{elem}, eof] & [List \rightarrow \underline{elem} List \bullet, eof] \\
 [List \rightarrow \underline{elem} \bullet, eof] & [List \rightarrow \bullet \underline{elem} List, \underline{elem}] \\
 [List \rightarrow \bullet \underline{elem}, \underline{elem}] & [List \rightarrow \underline{elem} \bullet List, \underline{elem}] \\
 [List \rightarrow \underline{elem} \bullet, \underline{elem}] & [List \rightarrow \underline{elem} List \bullet, \underline{elem}]
 \end{array}$$

Remarque : c'est la liste construite combinatoirement de tous les états possibles du parseur, mais parmi ces *items*, certains sont inaccessibles par une réduction d'une phrase valide de la grammaire, par exemple le dernier : lorsqu'on a reconnu une *List* et qu'il reste un *elem*, c'est une erreur.

Groupons à présent ces *items* en ensembles qui correspondent à un même état du parseur. Pour cela, prenons l'unique règle indiquant le but à réaliser pour reconnaître une phrase du langage (axiome). Elle nous donne l'*item*  $[But \rightarrow \bullet List, eof]$ , qui sert à initialiser l'état initial  $cc_0$  du parseur. Dans le même état, il faut rajouter les *item* qui peuvent être rencontrés pour réaliser ce premier *item*. On utilise pour cela la procédure *closure* :

*closure(s)*

tant que ( $s$  change)

pour chaque *item*  $[A \rightarrow \beta \bullet C\delta, a] \in s$

pour chaque production  $C \rightarrow \gamma \in P$

si  $\gamma = \epsilon$

$s \leftarrow s \cup \{[A \rightarrow \beta C \bullet \delta, a]\}$

sinon, pour chaque  $b \in First(\delta a)$

$s \leftarrow s \cup \{[C \rightarrow \bullet \gamma, b]\}$

Voici l'explication en français : si  $[A \rightarrow \beta \bullet C\delta, a] \in s$  alors une complétion possible du contexte déjà lu est de trouver une chaîne qui se réduit à  $C$  suivi de  $\delta a$ . Si cette réduction (par exemple  $C \rightarrow \gamma$ ) a lieu, on va donc rencontrer  $\gamma$  puis un élément de  $First(\delta a)$  juste après donc on passera forcément par un des *item* ajoutés dans la procédure *closure*.

Dans notre exemple, on a initialement  $[But \rightarrow \bullet List, eof]$  dans  $cc_0$ , le passage par la procédure *closure()* ajoute deux *items* :  $[List \rightarrow \bullet \underline{elem} List, eof]$  et  $[List \rightarrow \bullet \underline{elem}, eof]$ , comme les  $\bullet$  précèdent uniquement des non terminaux, cela ne génère pas plus d'*item* :

$$cc_0 = closure(s_0) = \{[But \rightarrow \bullet List, eof], [List \rightarrow \bullet \underline{elem} List, eof], [List \rightarrow \bullet \underline{elem}, eof]\}$$

**Table Goto** On construit une table qui prévoit l'état que le parseur devra prendre s'il est dans un certain état  $s = cc_i$  et qu'il a au sommet de la pile un symbole  $x \in T \cup NT$ .

```
goto(s, x)
  nextState ← ∅
  pour chaque item  $i \in s$ 
    si  $i$  est  $[A \rightarrow \beta \bullet x\delta, a]$  alors
      nextState ← nextState ∪  $\{[A \rightarrow \beta x \bullet \delta, a]\}$ 
  return closure(nextState)
```

C'est là qu'on commence à voir pourquoi on a besoin de la fonction closure : pour que la fonction goto fonctionne lorsque  $x$  est un terminal, on a besoin d'avoir dans l'état  $s$  tous les items dans lesquels on a "expansé" les non-terminaux à droite du  $\bullet$ . Toutefois, insistons sur le fait que la fonction goto est définie aussi sur les non-terminaux : la table Goto(s,x) de l'algorithme du parser fait une transition à partir de l'état  $s$  parfois le token de  $T$  en entrée (*shift*), parfois le symbole de  $NT$  au sommet de la pile (*reduce*).

Sur notre exemple, goto( $cc_0, \underline{elem}$ ) donne d'abord : les deux items :  $\{[List \rightarrow \underline{elem} \bullet List, eof], [List \rightarrow \underline{elem} \bullet, eof]\}$  puis la fermeture rajoute :

$$cc_2 = \text{goto}(cc_0, \underline{elem}) = \left\{ \begin{array}{l} [List \rightarrow \underline{elem} \bullet List, eof], [List \rightarrow \underline{elem} \bullet, eof], \\ [List \rightarrow \bullet \underline{elem} List, eof], [List \rightarrow \bullet \underline{elem}, eof] \end{array} \right\}$$

On a aussi

$$cc_1 = \text{goto}(cc_0, List) = \{[But \rightarrow List \bullet, eof]\}$$

et

$$cc_3 = \text{goto}(cc_2, List) = \{[List \rightarrow \underline{elem} List \bullet, eof]\}$$

et enfin : goto( $cc_2, \underline{elem}$ ) =  $cc_2$ .

### Collection canonique d'ensemble d'item LR(1)

```
cc0 ← closure(s0)
CC ← {cc0}
tant que (CC change)
  pour chaque ccj ∈ CC non marqué
    marquer ccj
    pour chaque x suivant un • dans un item de ccj
      temp ← goto(ccj, x)
      si temp ∉ CC
        alors CC ← CC ∪ {temp}
    enregistrer la transition de ccj à temp sur x
```

L'ensemble  $CC$  des états générés ainsi que les transitions enregistrées donnent directement la table Goto (on utilise que les transitions sur les non-terminaux pour la table Goto). Pour la table Action, on regarde les états générés. Il y a plusieurs cas :

1. Un item de la forme  $[A \rightarrow \beta \bullet \underline{c}\gamma, a]$  indique que le fait de rencontrer le terminal  $\underline{c}$  serait une étape valide pour la reconnaissance du non terminal  $A$ . L'action à générer est un shift sur  $\underline{c}$  dans l'état courant. Le prochain état est l'état généré avec la table Goto ( $\beta$  et  $\gamma$  peuvent être  $\epsilon$ ).
2. Un item de la forme  $[A \rightarrow \beta \bullet, a]$  indique que le parser a reconnu  $\beta$ ; si le symbole à lire est  $\underline{a}$  alors l'item est une réduction candidate, on génère donc un *reduce*  $A \rightarrow \beta$  sur  $\underline{a}$  dans l'état courant.
3. L'item  $[But \rightarrow S \bullet, eof]$  est unique ; il génère une acceptation.

Notez qu'on ignore, pour la table *Action*, les item où le  $\bullet$  précède un non-terminal. En fait, le fait que l'on ait exécuté la procédure closure assure que les items qui permettent de réduire ce non-terminal sont présents dans le même état.

Voici les deux tables pour notre exemple :

etat	<u>eof</u>	<u>elem</u>
cc <sub>0</sub>		shift cc <sub>2</sub>
cc <sub>1</sub>	accept	
cc <sub>2</sub>	reduce List → <u>elem</u>	shift cc <sub>2</sub>
cc <sub>3</sub>	reduce List → <u>elem</u> List	

etat	List	<u>eof</u>	<u>elem</u>
cc <sub>0</sub>	cc <sub>1</sub>		cc <sub>2</sub>
cc <sub>1</sub>			
cc <sub>2</sub>	cc <sub>3</sub>		cc <sub>2</sub>
cc <sub>3</sub>			

**Grammaire ambiguë** Si une grammaire n'est pas  $LR(1)$  (si elle est ambiguë, par exemple) alors cette propriété va apparaître lors de la construction des tables. On va se retrouver avec un conflit *shift – reduce* : un état contient à la fois les items  $[A \rightarrow \alpha \bullet \underline{c}\gamma, \underline{a}]$  et  $[B \rightarrow \beta \bullet, \underline{c}]$ . Il peut aussi contenir un conflit *reduce – reduce* : un état contient  $[A \rightarrow \alpha \bullet, \underline{a}]$  et  $[B \rightarrow \beta \bullet, \underline{a}]$ .

La meilleure méthode pour déterminer si une grammaire est  $LR(1)$  est de lancer un constructeur de parseur  $LR(1)$  dessus.

### 3.4 Quelques conseils pratiques

Bien que les parseurs soient maintenant essentiellement générés automatiquement, un certain nombre de choix sont laissés au programmeur.

**Résolution des conflits** Si vous utilisez `yacc` ou `bison` (le dernier étant la version GNU plus moderne et produisant du C ANSI du premier), vos premiers essais vont probablement produire des messages de *warning* du type "shift-reduce conflict" et "reduce-reduce conflict". Il est préférable de les comprendre et de reformuler la grammaire pour qu'ils disparaissent. Toutefois, ce sont des warnings et non pas des erreurs, parce que l'outil construit un parseur qui résout les conflits en utilisant des règles simples :

- En cas de conflit *shift/reduce*, il choisit l'alternative qui produit la phrase la plus longue, autrement dit il favorise le shift. Un avantage de cette stratégie est que s'il y a plusieurs conflits *shift/reduce* dans le même état, cela les résout tous (puisque'il n'y a qu'un shift possible). Notez que `yacc/bison` permet de définir une notion de précedence entre les token qui permet également de résoudre certains conflits. Voir la doc.
- En cas de conflit *reduce/reduce* l'outil favorisera la réduction qui correspond à la production écrite en premier dans le texte de la grammaire. Connaître cette règle permet de savoir ce qui se passe en présence de tels avertissements.

**Construction de l'arbre** On ne veut pas juste reconnaître le langage, on veut aussi produire une représentation sous forme d'arbre étiqueté. Le code réalisant cette construction s'insère soit avant les appels récursifs d'un parseur descendant, soit dans le cas d'une réduction pour un parseur ascendant. Les générateurs de parseurs comme `Yacc` permettent l'insertion de code. Si ce code manipule des variables globales, il faut être conscient qu'il s'exécutera dans l'ordre des réductions, qui n'est pas toujours trivial.

**Gestion d'erreur** Le parseur est largement utilisé pour déboguer les erreurs de syntaxe, il doit donc en détecter le maximum d'un seul coup. Pour l'instant on a toujours supposé que le parseur s'arrêtait lorsqu'il rencontrait une erreur. Si l'on désire continuer le parsing, il faut un mécanisme qui permette de remettre le parseur dans un état à partir duquel il peut continuer le parsing. Une manière de réaliser cela est de repérer des *tokens de synchronisation* qui permettent de synchroniser l'entrée lue avec l'état interne du parseur. Lorsque le parseur rencontre des erreurs, il oublie les tokens jusqu'à ce qu'il rencontre un token de synchronisation. Par exemple, dans beaucoup de langage "à la Algol" le point-virgule est un séparateur qui peut être utilisé comme token de synchronisation.

Pour les parseurs descendants, ça ne pose pas de problème. Pour les parseur  $LR(1)$ , on cherche dans la pile le dernier  $Goto[s, Statement]$  (i.e. celui qui a provoqué l'erreur), on dépile, avale les tokens jusqu'au début du prochain  $Statement$  et empile un nouveau  $Goto[s, Statement]$ . On peut indiquer aux générateurs de parseurs comment se synchroniser avec les règles de recouvrement d'erreurs (on indique dans les parties droites de règles où et comment ils peuvent se resynchroniser).

**Ambiguïté sensible au contexte** On utilise souvent les parenthèses pour représenter à la fois les accès aux tableaux et les appels de fonctions. Une grammaire permettant cela sera forcément ambiguë. Dans un parser descendant on peut tester le type de l'identificateur précédant les parenthèse. Pour les parseurs  $LR(1)$  on peut introduire un non terminal commun pour les deux usages (et tester le type de l'identificateur) ou réserver des identificateurs différents pour les tableaux et les fonctions.

**Récursion à droite ou à gauche** Les parsers  $LR(1)$  autorisent les grammaires récursives à gauche. On a donc le choix entre grammaires récursives à droite et à gauche. Pour décrire le langage, plusieurs facteurs entrent en ligne de compte :

- la taille de la pile. En général la récursion à gauche entraîne des piles de taille inférieure. La récursion à gauche va réduire plus rapidement et limiter la taille de la pile (à l'extrême : une liste de taille  $l$  peut entraîner une pile de taille 2 en récursif à gauche et  $l$  en récursif à droite).
- Associativité. La récursion à gauche ou à droite de la grammaire va imposer l'ordre dans lequel l'associativité va être faite (les grammaires récursives à gauche produisent de l'associativité à gauche si on a  $List \rightarrow List \text{ elem}$  on produit un AST qui descend à gauche et donc :  $((elem_1 \text{ elem}_2) \text{ elem}_3) \dots$  Même pour des opérateurs théoriquement associatifs cet ordre peut changer les valeurs à l'exécution. On peut éventuellement modifier l'associativité par défaut dans le parseur en manipulant explicitement l'AST

**Variations sur LR(1)** Il existe d'autres algorithmes de construction de parseur qui produisent des tables plus petites que la construction  $LR(1)$  classique. L'algorithme  $SLR(1)$  (Simple  $LR(1)$ ) impose à la grammaire que le symbole à lire en avant ne soit pas nécessaire pour effectuer une réduction par le bas. Les items sont alors de la forme  $\langle A \rightarrow \alpha \bullet \gamma \rangle$  sans le symbole censé suivre après  $\gamma$ . L'algorithme est similaire à  $LR(1)$  mais il décide de faire une réduction si le symbole suivant est dans l'ensemble  $Follow(A)$  entier (indépendant du contexte), là où  $LR(1)$  ne ferait la réduction que pour une seule valeur du symbole suivant.

Pour comprendre intuitivement pourquoi cela marche souvent, il suffit de considérer la grammaire des expressions. Les terminaux  $+$  et  $-$  jouent exactement le même rôle syntaxique : il paraît stupide de ne pas mettre les items  $[\alpha \bullet, +]$  et  $[\alpha \bullet, -]$  dans le même sac (pardon, état). Attention, il ne s'agit pas d'ignorer le token en question : on en a bien besoin pour savoir quel arbre on construit.

L'algorithme  $LALR(1)$  (Lookahead  $LR(1)$ ) rassemble certains items qui ne diffèrent que par le symbole suivant (items ayant le même *core*, le même item SLR). Cela produit une collection canonique semblable à celle de la construction  $LR(1)$  mais plus compacte.

Une grammaire  $LR(1)$  peut ne pas être  $LALR(1)$  (on peut montrer que tout langage ayant une grammaire  $LR(1)$  a aussi une grammaire  $SLR(1)$  et une grammaire  $LALR(1)$ ). Malgré la moindre expressivité, les parseurs  $LALR(1)$  sont extrêmement populaires du fait de leur efficacité (*Bison/Yacc* en est un). En fait pour une grammaire de langage quelconque il très probable que la transformation  $LR(1) \rightarrow LALR(1)$  de l'automate donne un automate sans conflit.

**Optimisations** La forme de la grammaire a un impact direct sur les performances de son analyse syntaxique (bien que la complexité asymptotique ne change pas). On peut réécrire la grammaire pour réduire la hauteur de l'arbre de syntaxe. Si l'on reprend la grammaire des expressions classiques et que l'on déroule le non-terminal *Factor* dans l'expression de *Term* on obtient 9 parties droites possibles pour *Term* mais on réduit la hauteur générale de tout arbre d'expression arithmétique de 1. En parsing  $LR(1)$  sur l'expression  $x - 2 \times y$ , cela élimine trois réductions sur

6 (le nombre de shift reste bien sûr inchangé – un par token). En général on peut éliminer toutes les productions “inutiles”, c’est à dire ne comportant qu’un seul symbole en partie droite. En revanche la grammaire résultante est moins lisible et comme on va le voir plus loin, les actions que pourra réaliser le parseur seront moins faciles à contrôler. Cette transformation peut aussi augmenter la taille des tables générées (ici  $CC$  passe de 32 ensembles à 46 ensembles), le parseur résultant exécute moins de réductions mais manipule des tables plus grandes.

On peut aussi regrouper des non terminaux dans des catégories (par exemple, + et –) et ne mettre qu’une seule règle pour tous, la différenciation étant faite en testant explicitement la valeur du symbole à postériori.

Outres les techniques précédentes, qui sont à la charge du rédacteur de la grammaire, de gros efforts sont faits au niveau des outils pour réduire les tables des parseurs  $LR(1)$ . Lorsque les lignes ou les colonnes des tables sont identiques, le concepteur du parseur peut les combiner et renommer les lignes des tables en conséquence (une ligne peut alors représenter plusieurs états). Pour la grammaire des expressions simple, cela produit une réduction de taille de 28%. On peut aussi choisir de combiner deux lignes dont le comportement ne diffère que pour les erreurs, on acceptera peut-être alors des codes incorrects.

On peut aussi décider d’implémenter le contenu de *Action* et *Goto* directement en code plutôt que dans une table. chaque état devient alors un gros *case*. Cela devient difficile à lire mais peut être très efficace

### 3.5 Résumé sur l’analyse lexicale/syntaxique

- L’analyse lexicale transforme un flot de caractères en un flot de token. L’analyse syntaxique transforme un flot de token en arbre syntaxique.
- Un token est généralement une classe et une chaîne de caractère (son nom) (on peut quelquefois rajouter des informations sur sa position dans la chaîne).
- Les analyseurs lexicaux peuvent être générés à partir des automates d’états finis, eux-mêmes générés à partir des expressions régulières décrivant des langages.
- L’analyseur lexical fait un traitement lorsqu’il rencontre un identificateur (en général il le référence dans une table des symboles).
- L’analyse syntaxique peut se faire de manière descendante ou ascendante (se référant à la manière avec laquelle on construit l’arbre syntaxique).
- Les parseurs descendants écrits à la main consistent en un ensemble de procédures mutuellement récursives structurées comme la grammaire.
- Pour écrire ces parseurs, il faut avoir calculé les ensembles *First* et *Follow*. Les grammaires acceptées par ces parseurs sont dits  $LL(1)$ .
- Les grammaires récursives à gauche ne sont pas  $LL(1)$ , on peut dérécurser une grammaire récursive à gauche.
- Les parseurs ascendants manipulent des items : à la base c’est une règle avec un point (dot) symbolisant l’endroit jusqu’auquel on a reconnu (éventuellement associé à un ou plusieurs symboles représentant ce que l’on rencontrera après avoir fini de reconnaître la règle en question).
- Les parseurs ascendants essaient d’identifier successivement les réductions candidates (*handle*). Il y a de nombreuses techniques pour trouver les réductions candidates. Les parseurs  $LR(1)$  utilisent des ensembles d’items  $LR(1)$ .
- Un ensemble d’items représente un état dans lequel peut se trouver le parseur  $LR(1)$  à un moment donné. Formellement, les états de l’automate à partir duquel est construit le parseur  $LR(1)$  sont des ensembles d’items.
- En parsing  $LR(0)$  tout item de la forme  $[N \rightarrow \alpha \bullet]$  génère une réduction. En parsing  $SLR(1)$  un item  $[N \rightarrow \alpha \bullet]$  génère une réduction si et seulement si le prochain symbole à lire appartient à  $Follow(N)$ . En parsing  $LR(1)$  un item  $[N \rightarrow \alpha \bullet, A]$  génère une réduction si le prochain symbole à lire est dans l’ensemble  $A$ , un petit ensemble de terminaux. En  $LR(1)$   $A$  est limité à un symbole. En  $LALR(1)$  c’est un ensemble de symboles (dans  $Follow(N)$ ). En pratique une grammaire  $LR(1)$  a de très fortes chances d’être  $LALR(1)$ .
- Concernant les classes de grammaires, on a les relations :  $RG \subset LL(1) \subset LR(1) \subset CFG$ . L’ambiguïté entraîne le fait que la grammaire n’est pas  $LR(1)$ , mais ce n’est pas le seul cas.

Étant donnée une grammaire, on ne sait pas dire si on peut trouver une autre grammaire équivalente  $LR(1)$ . La méthode la plus simple pour savoir si une grammaire est  $LR(1)$  est de construire le parseur et de voir s'il y a des conflits.

## 4 Analyse sensible au contexte

On a vu que l'on pouvait introduire certaines informations sémantiques dans la phase de parsing. Mais cela ne suffit pas. On a souvent besoin de plus d'information que l'on ne peut coder avec une grammaire hors contexte.

Par exemple, lorsqu'on rencontre un identificateur, est-ce une fonction ou une variable ? Si c'est une variable, est-elle déclarée ? Est-elle utilisable à cet endroit (dans ce contexte) ? Quel type de valeur est stocké dans cette variable ? Quelle place prend-elle en mémoire ? Faut-il allouer de l'espace mémoire pour elle ? Si c'est une fonction, combien et quel type d'argument a-t-elle ? Toutes ces informations sont disponibles dans le programme mais pas forcément à l'endroit où l'on rencontre l'identificateur : on dit qu'elles sont présentes dans le *contexte*, et on les appellera informations *contextuelles*.

En plus de reconnaître si le programme est bien une phrase du langage, le parseur doit donc construire ce contexte, qui lui permettra ensuite d'avoir accès à ces informations en temps voulu. Le contexte est défini par des déclarations (en général en début de programme/fonction/procédure...) qui donnent le contexte pour les feuilles de l'arbre syntaxique, et par des règles d'inférences qui permettent de déduire le contexte pour tous les nœuds internes de l'arbre.

### 4.1 Introduction aux systèmes de type

La plupart des langages propose un système de type c'est à dire un ensemble de propriétés associées à chaque valeur. La théorie des types a généré énormément de recherches et de nouveaux concepts de programmation que nous ne mentionneront pas ici. Pour simplifier, les types ont été historiquement des bricolages ad-hoc ajoutés après coup aux langages de programmation séquentiels, mais ils ont été au cœur de la définition du paradigme fonctionnel, jusqu'à Curry-Howard qui dit en gros qu'un bon typage d'un programme est équivalent à une preuve de ce programme. Il y a encore en 2005 des gens qui font de la recherche sur (au hasard) Caml, et ils font de la recherche sur les systèmes de type. Ceux qui font de la recherche sur Java, d'ailleurs, c'est pareil (ils cherchent des failles de sécurité dedans, par exemple).

Le but d'un système de typage est de spécifier le comportement du programme plus précisément qu'avec une simple grammaire hors contexte. Le système de type crée un deuxième vocabulaire pour décrire la forme et le comportement des programmes valides. Il est essentiellement utile pour :

- la sûreté à l'exécution : il permet de détecter à la compilation beaucoup plus de programmes mal formés.
- augmenter l'expressivité : il permet d'exprimer au moins plus naturellement certains concepts (par exemple par la surcharge d'opérateurs élémentaires comme l'addition).
- générer un meilleur code : il permet d'enlever des tests à l'exécution puisque le résultat est connu à la compilation (exemple : comment stocker une variable quand on ne connaît pas son type).

#### 4.1.1 Composant d'un système de typage

En général tous les langages ont un système de type basé sur :

**Des types de base** En général entiers de différentes tailles, flottants idem, caractères itou, et booléens (quelquefois les chaîne de caractères).

Petit jeu : Soit la liste suivante de (quelques) types de base du langage C : `char`, `unsigned char`, `int`, `long int`, `long long int`, `float`, `double`, `long double`. En déduire que ses concepteurs n'avaient jamais entendu parler de la notion de sémantique.

**Des constructeurs de types** À partir de ces types de base, on peut construire des structures de données plus complexes pour représenter les graphes, arbres, tableaux, listes, piles, etc. Parmi les plus fréquents on trouve : les tableaux avec des implémentations diverses (ordre de linéarisation,



contraintes sur les bornes) et éventuellement des opérations associées. Les chaînes de caractères sont quelquefois implémentées comme des tableaux de caractères.

Le produit cartésien de type :  $type_1 \times type_2$  (souvent appelé structure) permet d'associer plusieurs objets de type arbitraire. L'union de deux types  $type_1 \cup type_2$  définit le type "de  $type_1$  ou de  $type_2$ ". Les ensembles énumérés sont souvent autorisés, et enfin les pointeurs permettent d'avoir une certaine liberté quant à la manipulation de la mémoire.

**Équivalence de types** On peut décider que deux types sont dits équivalents lorsqu'ils ont le même nom (plus strict, plus sûr et plus expressif, mais aussi plus pénible pour le programmeur) ou lorsqu'ils ont la même structure.

Toute la question est de savoir si on a le droit d'additionner des pommes et des oranges. Les langages objets permettent de dire qu'on n'a pas le droit, mais que les pommes et les oranges sont des fruits et qu'on a le droit d'additionner des fruits.

**Des règles d'inférence** Les règles d'inférence spécifient pour chaque opérateur les relations entre le type des opérandes et le type du résultat. Elles permettent d'affecter un type à toutes les expressions du langage et de détecter des erreurs de type.

**Inférence de type pour toutes les expressions** Souvent les langages imposent une règle de déclaration de type avant utilisation. Dans ce cas, l'inférence et la détection d'erreur de type peut se faire à la compilation. Sinon (ou pour les constantes par exemple) le type est choisi lorsqu'on rencontre l'expression, le mécanisme mis en place est plus complexe. Le système doit forcément contenir des règles pour l'analyse interprocédurale et définir des signatures (types des arguments et résultats) pour chaque fonction.

#### 4.1.2 Remarques

Il y a des langages fortement typés (toute expression du langage peut être typée), et des langages faiblement typés (il existe des expressions qui ont plusieurs types valide, et débrouille-toi). Dans les langages fortement typés, certains peuvent avoir la vérification de type entièrement réalisée à la compilation (*statically strongly typed*) et d'autres dont la vérification de type est (partiellement) faite à l'exécution (*dynamically*). Pour ces derniers, le compilateur n'est pas plus simple pour autant : il doit générer le code qui fera la vérification...

Fortement typé ne veut pas dire qu'on est obligé de déclarer toutes les variables. Il existe des algorithmes d'*unification* pour trouver le type le plus large compatible avec l'utilisation qui est faite des variables.

Les langages fournissent également en général des opérateurs explicites de *coercion* (ou promotion, ou *cast*) d'un type en un autre. Le compilateur peut également insérer automatiquement des cast implicites dans l'AST. Par exemple, en C  $1+1.0$  provoque la coercion implicite de 1 (entier) en un flottant, et l'addition est l'addition flottante. En Caml, il faut avoir transformé l'entier explicitement en un flottant, et utiliser comme il faut soit l'opérateur entier  $+$ , soit l'opérateur flottant  $+. .$

Java est fortement typé et pourrait avoir ses types vérifiés statiquement, sauf que le modèle d'exécution interdit au compilateur de voir l'ensemble du programme à la fois. Du coup, il faut faire un peu de vérification dynamique des types.

## 4.2 Les grammaires à attributs

Les grammaires à attributs ont été proposées pour résoudre certains problèmes de transmission d'information dans l'arbre de syntaxe. La grammaire hors contexte du langage est fournie par le concepteur du langage ainsi que la sémantique associée. Le concepteur du compilateur introduit alors un ensemble d'attributs pour les terminaux et non terminaux du langage et un ensemble de règles associées aux règles de la grammaire qui permet de calculer ces attributs.

Considérons la grammaire  $SBN = (T, NT, S, P)$  suivante représentant les nombres binaires signés :

$$T = \{+, -, 0, 1\} \quad NT = \{Number, Sign, List, Bit\} \quad S = \{Number\}$$

$$P = \left\{ \begin{array}{l} \text{Number} \rightarrow \text{Sign List} \\ \text{Sign} \rightarrow + \\ \quad \quad \quad | \\ \quad \quad \quad - \\ \text{List} \rightarrow \text{List Bit} \\ \quad \quad \quad | \\ \quad \quad \quad \text{Bit} \\ \text{Bit} \rightarrow 0 \\ \quad \quad \quad | \\ \quad \quad \quad 1 \end{array} \right\}$$

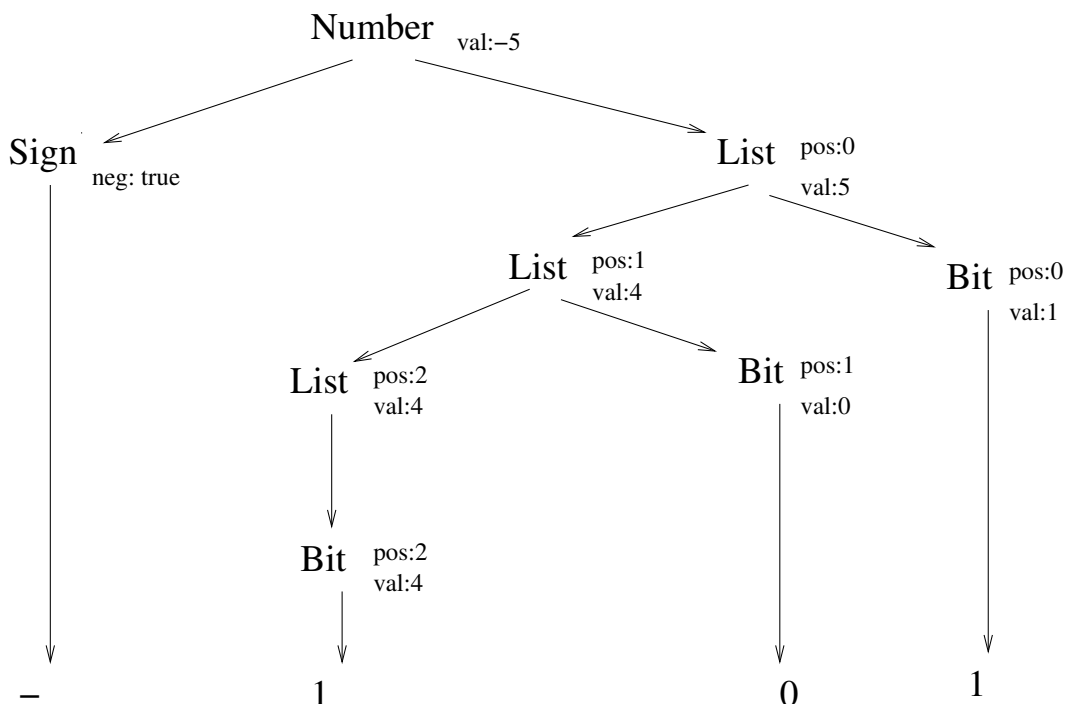
On va annoter cette grammaire avec des attributs permettant de calculer la valeur du nombre binaire signé qui est représenté :

Symbole	Attribut
Number	val
Sign	neg
List	pos, val
Bit	pos, val

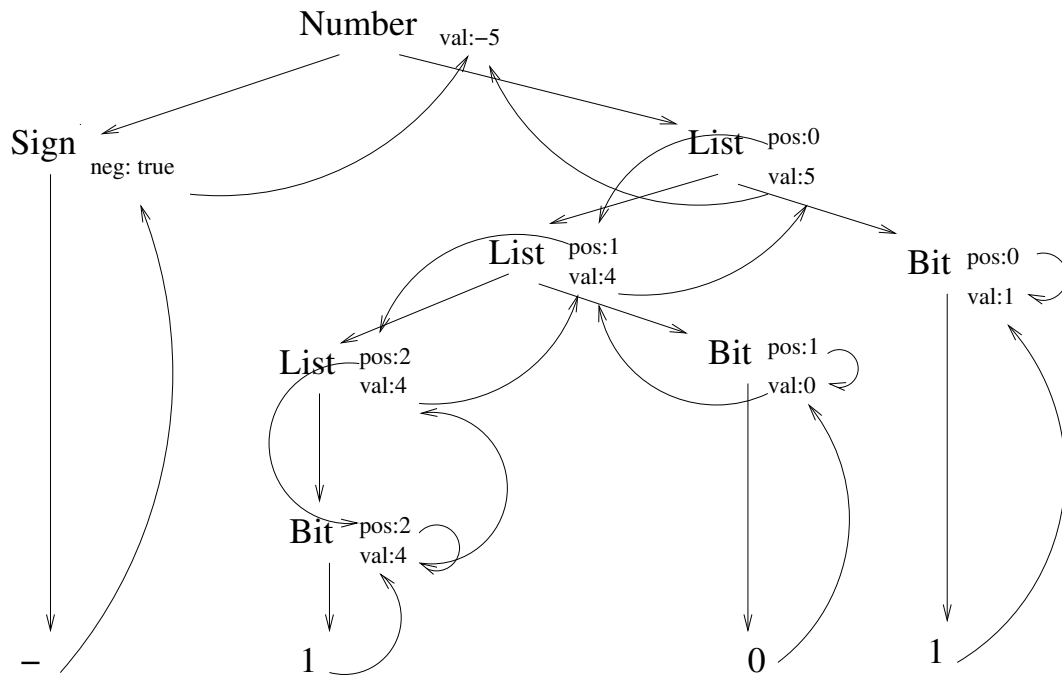
On va ensuite définir les règles pour définir les attributs de chaque symbole. Ces règles sont associées aux règles de la grammaire :

1.	$\text{Number} \rightarrow \text{Sign List}$	$\text{List.pos} \leftarrow 0$ <i>si</i> $\text{Sign.neg}$ <i>alors</i> $\text{Number.val} \leftarrow -\text{List.val}$ <i>sinon</i> $\text{Number.val} \leftarrow \text{List.val}$
2.	$\text{Sign} \rightarrow +$	$\text{Sign.neg} \leftarrow \text{false}$
3.	$\text{Sign} \rightarrow -$	$\text{Sign.neg} \leftarrow \text{true}$
4.	$\text{List} \rightarrow \text{Bit}$	$\text{Bit.pos} \leftarrow \text{List.pos}$ $\text{List.val} \leftarrow \text{Bit.val}$
5.	$\text{List}_0 \rightarrow \text{List}_1 \text{Bit}$	$\text{List}_1.pos \leftarrow \text{List}_0.pos + 1$ $\text{Bit.pos} \leftarrow \text{List}_0.pos$ $\text{List}_0.val \leftarrow \text{List}_1.val + \text{Bit.val}$
6.	$\text{Bit} \rightarrow 0$	$\text{Bit.val} \leftarrow 0$
7.	$\text{Bit} \rightarrow 1$	$\text{Bit.val} \leftarrow 2^{\text{Bit.pos}}$

Par exemple, pour la chaîne  $-101$ , on va construire l'arbre suivant :



On peut représenter les dépendances entre les attributs :



Pour construire cet arbre, on a exécuté un parseur qui a créé une instance des attributs pour chaque noeud et on a ensuite exécuté un *évaluateur* qui a calculé les valeurs de chaque attribut.

Les règles sur les attributs induisent des dépendances implicites. Selon la direction des dépendances entre les attributs, on parle d'attributs *synthétisés* (lorsque le flot des valeurs va de bas en haut) ou *hérités* (lorsque le flot des valeurs va de haut en bas, ou lorsque les valeurs viennent de lui même ou de ses frères). Il faut qu'il n'y ait pas de circuits dans un tel graphe de dépendance. Alors, le problème de l'évaluation des attributs est un problème de point fixe.

Pour les grammaires non circulaires, il existe des générateurs d'évaluateurs, qui étant donnée une grammaire à attributs, vont générer un évaluateur pour les attributs. On a une certaine liberté pour l'ordre d'évaluation du moment qu'il respecte les dépendances. On peut utiliser des méthodes dynamiques (qui maintiennent une liste d'attributs prêts à être évalué), des méthodes systématiques (parcours systématique de l'arbre jusqu'à convergence), des méthodes basées sur les règles (on analyse statiquement l'ordre dans lequel peuvent être exécutées les règles, voire on crée une méthode dédiée à certains attributs).

**Limitations des grammaires à attributs** Pour certaines informations qui se transmettent de proche en proche dans un seul sens, les grammaires à attributs sont bien adaptées. En revanche beaucoup de problèmes pratiques apparaissent lorsqu'on veut transmettre d'autre type d'informations.

- Information non locale. Si l'on veut transmettre une information non locale par des attributs (e.g. le type d'une variable stockée dans une table), on doit ajouter des règles pour tous les non terminaux afin de transporter cette information jusqu'à l'endroit où elle est utile. C'est lourdingue, et il faut maintenir tout cela quand on modifie la grammaire.
- Place mémoire. Sur des exemples réalistes, l'évaluation produit un très grand nombre d'attributs. Le caractère fonctionnel de la transmission des attributs peut entraîner une place utilisée trop grande, or on ne sait pas a priori quand est ce que l'on va pouvoir désallouer cette mémoire.
- Utilisation de l'arbre de parsing. Les attributs sont instanciés sur les noeuds de l'arbre de parsing (i.e. l'arbre représentant exactement la dérivation dans la grammaire). Comme on va le voir, peu de compilateurs utilisent véritablement l'arbre syntaxique; on utilise en général un arbre de syntaxe abstrait (*abstract syntax tree*, AST) qui est une simplification de l'arbre syntaxique. Il peut arriver qu'on utilise pas du tout de représentation sous forme d'arbre, le coût de la gestion additionnelle d'un arbre doit être évalué.

- Accès aux informations. Les informations récupérées par les attributs sont disséminées dans l'arbre. L'accès aux informations ne se fait pas en temps constant. Ce problème peut être résolu par l'utilisation de variables globales stockant les attributs. Dans ce cas, les dépendances entre attributs ne sont plus explicites mais par les accès aux cases du tableau (le paradigme n'est plus fonctionnel)

Pour toutes ces raisons, les grammaires à attributs n'ont pas eu le succès qu'ont eu les grammaires hors contexte pour la définition de compilateurs. En général, une fois qu'on a obtenu l'AST, il est plus simple de l'étiqueter "à la main" en écrivant le parcours qui va bien. Elles restent néanmoins un moyen relativement propre et efficace de calculer certaines valeurs lorsque la structure du calcul s'y prête bien.

### 4.3 Les grammaires contextuelles

Rappel : ce sont les grammaires dans lesquelles les productions sont du type  $\alpha A \beta \rightarrow \alpha \gamma \beta$ , où  $A \in NT$  et  $\alpha, \beta$  et  $\gamma$  sont dans  $(T \cup NT)^*$ . Là dedans,  $\alpha$  et  $\beta$  représentent le contexte dans lequel on peut réduire  $A$  en  $\gamma$ .

Personne ne s'en est jamais servi pour faire des compilateurs, pour plusieurs raisons.

- Les parseurs ont une complexité bien plus mauvaise que les parseurs des grammaires hors-contexte : temps cubique en la taille de l'entrée, espace PSpace-complet (ce qui signifie en gros : ce qui se fait de pire en restant polynomial).
- Certaines notions sémantiques (par exemple, qu'une variable doit être déclarée avant d'être utilisée) ne s'expriment pas mieux dans une grammaire contextuelle que dans une grammaire hors contexte : le "contexte" au sens de la grammaire contextuelle est un contexte local à une production, alors que le contexte dont on a besoin dans cet exemple est un contexte global. Bien sûr on peut exprimer ce contexte global de manière syntaxique par des techniques d'énumération, mais l'explosion combinatoire rend cette technique inutilisable en pratique.
- On peut ressortir l'argument habituel selon lequel les grammaires hors-contextes suffisent à tous les langages dont on a besoin, donc pourquoi s'embêter ? C'est peut-être historique (on a pris l'habitude de construire les langages comme cela) mais c'est un fait.

Et donc fini les tentatives d'approches théoriques. On va en rester aux grammaires hors-contexte sans attributs ou presque, et bricoler par dessus pour y ajouter la sémantique du langage.

### 4.4 Traduction ad-hoc dirigée par la syntaxe

L'idée sous-jacente aux grammaires à attributs est de spécifier une séquence d'actions associée aux règles. Ce principe est maintenant implémenté dans tous les compilateurs mais sous une forme moins systématique que dans la théorie des grammaires à attributs.

Il n'y aura qu'un seul attribut par noeud (qui peut être considéré comme la *valeur* associée au noeud) et cet attribut sera systématiquement synthétisé (calculé de bas en haut) lors des réductions. Les actions seront des bouts de code qui décrivent comment réaliser cette synthèse.

Plus précisément, le concepteur du compilateur ajoute à chaque production de la grammaire un morceau de code ou *action* qui sera exécutés lors de l'analyse syntaxique. Chaque fois que le parseur reconnaît cette production, il exécute l'action. Dans un parser descendant, il suffit d'ajouter le code dans les procédures associées à chaque règle (juste avant le retour). Dans un parseur LR(1) les actions sont à réaliser lors des réductions.

Lorsqu'il arrive aux feuilles (identificateurs ou constantes), le parseur fait référence à la table des symboles construite lors du parsing des déclarations. La table des symboles (souvent implémentée comme une table de hachage) peut contenir divers types d'informations comme le type, la taille de la représentation à l'exécution, les informations nécessaires pour générer l'adresse à l'exécution, la signature pour les fonctions, les dimensions et bornes pour les tableaux etc.

En fait, on pourra donc tout de même hériter des attributs, à condition que, comme la table des symboles,

- ce soient des variables globales,
- et l'ordre des réductions (LR(1) ou LL(1)) les ait définis avant qu'on les utilise.

Par exemple, voici une manière simple de réaliser le calcul de la valeur représentée par un mot de la grammaire *SBN* :

1.	<i>Number</i>	$\rightarrow$	<i>Sign List</i>	<i>Number.val</i>	$\leftarrow$	<i>Sign.val</i> $\times$ <i>List.val</i>
2.	<i>Sign</i>	$\rightarrow$	+	<i>Sign.val</i>	$\leftarrow$	1
3.	<i>Sign</i>	$\rightarrow$	-	<i>Sign.val</i>	$\leftarrow$	-1
4.	<i>List</i>	$\rightarrow$	<i>Bit</i>	<i>List.val</i>	$\leftarrow$	<i>Bit.val</i>
5.	<i>List</i> <sub>0</sub>	$\rightarrow$	<i>List</i> <sub>1</sub> <i>Bit</i>	<i>List</i> <sub>0</sub> .val	$\leftarrow$	2 $\times$ <i>List</i> <sub>1</sub> .val + <i>Bit.val</i>
6.	<i>Bit</i>	$\rightarrow$	0	<i>Bit.val</i>	$\leftarrow$	0
7.	<i>Bit</i>	$\rightarrow$	1	<i>Bit.val</i>	$\leftarrow$	1

**Implémentation** En pratique, il faut proposer un mécanisme pour passer les valeurs entre leur définition et leurs utilisations, en particulier un système de nommage cohérent. On peut utiliser la pile du parser *LR(1)* qui stockait déjà un couple  $\langle symbol, etat \rangle$  pour stocker maintenant un triplet  $\langle symbol, etat, valeur \rangle$ , dans le cas d'un attribut par symbole. Pour nommer ces valeurs, on va utiliser la notation de *Yacc* qui est très répandu.

Le symbol  $\$ \$$  réfère l'emplacement du résultat de la règle courante. Donc  $\$ \$ \leftarrow 0$  empilera 0 lorsque la réduction de la règle concernée aura lieu (en plus d'empiler l'état et le symbole). Les emplacements des symboles de la partie droite de la règle sont représentés par  $\$ 1, \$ 2 \dots \$ n$ . On peut réécrire l'exemple précédent avec ces conventions :

1.	<i>Number</i>	$\rightarrow$	<i>Sign List</i>	$\$ \$ \leftarrow$	$\$ 1 \times \$ 2$
2.	<i>Sign</i>	$\rightarrow$	+	$\$ \$ \leftarrow$	1
3.	<i>Sign</i>	$\rightarrow$	-	$\$ \$ \leftarrow$	-1
4.	<i>List</i>	$\rightarrow$	<i>Bit</i>	$\$ \$ \leftarrow$	$\$ 1$
5.	<i>List</i> <sub>0</sub>	$\rightarrow$	<i>List</i> <sub>1</sub> <i>Bit</i>	$\$ \$ \leftarrow$	2 $\times$ $\$ 1$ + $\$ 2$
6.	<i>Bit</i>	$\rightarrow$	0	$\$ \$ \leftarrow$	0
7.	<i>Bit</i>	$\rightarrow$	1	$\$ \$ \leftarrow$	1

**Actions intermédiaires** Lorsque le programmeur veut effectuer une action entre l'évaluation de deux symboles en partie droite d'une production, il peut couper cette partie droite en deux, et ajouter une production intermédiaire avec un nouveau non-terminal. De même, pour insérer une action lors d'un shift de *A*, il suffit d'ajouter une production "vide" de type  $A' \rightarrow A$ , et de remplacer toutes les occurrences de *A* par des *A'*.

**Arbre syntaxique complet et arbre syntaxique abstrait** En général on préfère une version compressée de l'arbre de syntaxe appelée *arbre de syntaxe abstraite* ou *AST*. Pour cela, on garde les éléments essentiels de l'arbre de syntaxe mais on enlève les noeuds internes qui ne servent à rien. Par exemple :

- Pour les règles inutiles du type  $Term \rightarrow Factor$ , l'action à réaliser est simplement de passer le pointeur sur le noeud de *Factor* directement à *Term* plutôt que de construire un nouveau noeud.
  - Les opérateurs sont représentés différemment, le type d'opérateur (par exemple + ou  $\times$ ) n'est plus une feuille mais un arbre binaire dont les feuilles sont les opérandes.
- On peut aussi transformer au vol des peignes droits en peignes gauches, etc.

**Construction d'un arbre typé** Cette approche permet aussi de construire un arbre lors du parsing et de vérifier les typages simultanément. Considérons par exemple la grammaire des expressions simples. On peut construire un arbre de syntaxe complet tout en évaluant le type de chaque noeud : en effet, ce typage est purement synthétisé.

**Mon premier générateur de code** En place d'un arbre compressé et typé, une traduction ad-hoc permet de générer directement du code assembleur à partir d'une expression arithmétique.

On suppose que l'on dispose des procédures *NextRegister* qui retourne un nom de registre non utilisé, et *Emit* qui génère le code correspondant à l'opération. L'information synthétisée par le

parseur est le registre qui contiendra le résultat, et les émissions de code sont des effets de bord. Vérifiez qu'elles ont lieu dans un ordre acceptable.

Production	Action
$Expr \rightarrow Expr \pm Term$	$$$ \leftarrow NextRegister$ $Emit(add \$1, \$3 \rightarrow \$\$)$
$Expr \rightarrow Expr \mp Term$	$$$ \leftarrow NextRegister$ $Emit(sub \$1, \$3 \rightarrow \$\$)$
$Expr \rightarrow Term$	$$$ \leftarrow \$1$
$Term \rightarrow Term * Factor$	$$$ \leftarrow NextRegister$ $Emit(mult \$1, \$3 \rightarrow \$\$)$
$Term \rightarrow Term / Factor$	$$$ \leftarrow NextRegister$ $Emit(div \$1, \$3 \rightarrow \$\$)$
$Term \rightarrow Factor$	$$$ \leftarrow \$1$
$Factor \rightarrow ( Expr )$	$$$ \leftarrow \$2$
$Factor \rightarrow \underline{number}$	$$$ \leftarrow NextRegister$ $Emit(loadI ValueFromLex \rightarrow \$\$)$

Bien sûr, tout ceci suppose qu'on a un nombre infini de registres. On y reviendra.

## 5 Représentations Intermédiaires

La structure de données utilisée pour effectuer les différentes passes de compilation après analyse syntaxique est appelée la représentation intermédiaire, en anglais IR. En fait il y aura plusieurs représentations intermédiaires pour les différentes passes du compilateur. On distingue deux grandes classes de représentations intermédiaires :

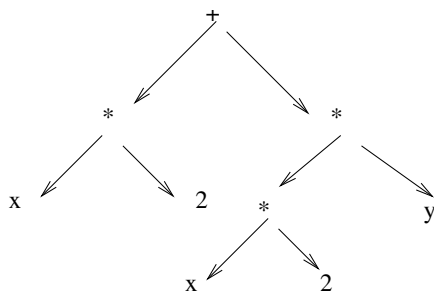
- Les représentations intermédiaires à base de graphes. Les algorithmes travaillant dessus manipulent des noeuds, des arcs, des listes, des arbres, etc. Les arbres de syntaxe abstraite construits au chapitre précédent sont de ce type.
- Les représentations intermédiaires linéaires qui ressemblent à du pseudo-code pour une machine abstraite. Les algorithmes travaillent sur de simple séquences linéaires d'opérations. La représentation ILOC introduite plus tôt en est un exemple.

En pratique de nombreuses représentations intermédiaires mélangent les deux notions (représentation hybrides, comme le graphe de flot contrôle). Ces représentations influencent fortement la manière dont le concepteur du compilateur va programmer ses algorithmes.

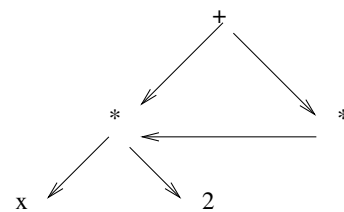
### 5.1 Graphes

#### 5.1.1 AST et DAG

On a vu la simplification de l'arbre de syntaxe en arbre de syntaxe abstrait. C'est l'outil idéal pour les traductions source à source. Un des problèmes des AST est qu'ils peuvent prendre une place relativement importante. On peut contracter un AST en un graphe acyclique direct (DAG) en regroupant les noeuds équivalents. Par exemple, on peut représenter l'expression  $x \times 2 + x \times 2 \times y$  par :



AST



DAG

Ce type d'optimisation doit être fait avec précaution ; on doit être sûr que la valeur de  $x$  ne change pas entre les deux utilisations. On peut utiliser pour cela un système de table de hachage basé sur la syntaxe externe des morceaux de code.

On utilise cette optimisation d'une part parce qu'elle réduit la place utilisée mais aussi parce qu'elle expose des redondances potentielles.

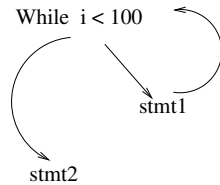
#### 5.1.2 CFG

Le *graphe de flot de contrôle* (*control flow graph*, CFG) modélise la manière dont le programme transfère le contrôle entre les blocs de code d'une procédure. Exemple de graphes de flot de contrôle :

```

While (i < 100)
  begin
    stmt1
  end
stmt2

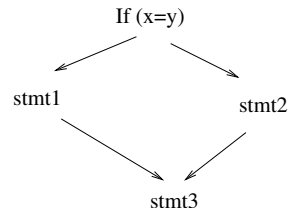
```



```

If (x = y)
  then stmt1
  else stmt2
stmt3

```



Ce graphe permet de suivre l'exécution qui aura lieu beaucoup mieux qu'avec l'AST. On a le choix de la granularité, c'est à dire de ce que contient un noeud du graphe. En général, un noeud sera un *bloc de base* (basic bloc). Un bloc de base représente une portion de code dans laquelle aucun branchement n'a lieu. En outre on impose souvent qu'il n'y ait qu'un seul point d'entrée pour un bloc de base.

Pour se rapprocher du code objet, on transformera en général les instructions de contrôle complexe comme *while*, *for* en leur version détaillée à base uniquement d'instruction arithmétiques, de comparaisons et de sauts conditionnels. Cela produit un CFG plus gros mais dont les noeuds sont moins variés, ce qui simplifie l'analyse.

Les blocs de base sont représentés en utilisant une autre représentation intermédiaire : AST ou code linéaire trois adresses.

### 5.1.3 Graphe d'appel

Le *call graph* est le graphe décrivant les appels (éventuellement récursifs) des procédures d'un programme. Il sert à définir les mécanismes de passage des paramètres et d'édition de lien (on verra à la section suivante), et éventuellement à faire de l'optimisation interprocédurale.

### 5.1.4 Graphe de dépendance

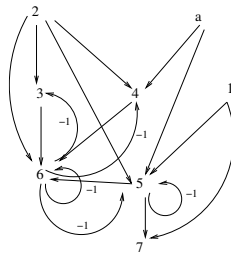
Le graphe de dépendance (*dependence graph*, DG) sert à représenter le flot des données (valeurs) entre les endroits où elles sont définies et les endroits où elles sont utilisées. Un noeud du graphe de dépendance représente une opération. Un arc connecte deux noeuds si l'un doit être exécuté avant l'autre. Il peut y avoir plusieurs cas de figure : lecture d'une variable après son écriture, écriture après une lecture ou deux écritures dans la même variable.

Le graphe de dépendance permet de savoir si on peut réordonner les calculs sans changer la sémantique du programme. Il est généralement utilisé temporairement pour une passe particulière puis détruit. L'exemple suivant montre un graphe de dépendance, les arcs étiquetés par -1 indique que la dépendance a lieu sur un instruction de l'itération précédente de la boucle :

```

1  x ← 0
2  i ← 1
3  while (i < 100)
4    if (a[i] > 0)
5      then x ← x + a[i]
6    i ← i + 1
7  print x

```



On peut vouloir exprimer l'information de dépendance de manière plus ou moins précise. L'analyse de dépendance devient complexe à partir du moment où l'on souhaite identifier les éléments des tableaux individuellement. Sans cette analyse précise, on est obligé de grouper toutes les instructions modifiant le tableau *a* dans un nom appelé *a*. Une technique pour raffiner cette analyse a été proposée par Paul Feautrier : *array data-flow analysis*.



## 5.2 IR linéaires

Le matériel fonctionnant de manière séquentielle, ces représentations intermédiaires sont utilisées pour manipuler le programme lorsqu'on se rapproche de la génération de code (l'assembleur est la dernière représentation intermédiaire linéaire utilisée par le compilateur). La différence principale entre l'assembleur cible et les IR linéaire utilisées au cours de la compilation est que ces dernières permettent en général un nombre infini de variables (l'allocation de registres sera une des dernières étapes de la compilation). Elles sont aussi en général plus simples : les instructions complexes et puissantes de la machine cible (modes d'adressages exotiques, instructions arithmétiques vectorielles, décalage sur le second opérande des ARM, etc) seront recherchées à la génération de code, ce qui permet d'avoir une représentation intermédiaire minimaliste et multi-cible.

Historiquement, on a utilisé des IR linéaires correspondant aux architectures cibles du moment : codes une adresse (machine à pile), codes à deux adresses (la destination est l'un des opérandes, qui est donc perdu) ou codes trois adresses (machines RISC).

### 5.2.1 Code à une adresse pour machine à pile

Les codes une adresse (ou code de machine à pile) suppose la présence d'une pile sur laquelle se font tous les calculs. Les opérandes sont lus dans la pile (et dépilés, on dit détruits) et le résultat est poussé sur la pile. Par exemple pour la soustraction de deux entiers on va exécuter  $push(pop() - pop())$ . Ce type de code est apparu pour programmer des machines à pile qui ont eu du succès jadis.

Plus récemment, un des avantages des codes une adresse est qu'ils sont relativement compacts (la pile crée un espace de nommage implicite et supprime de nombreux noms). C'est pourquoi on les utilise souvent pour le *bytecode* (langage machine) des machines virtuelles (Java, smalltalk).

### 5.2.2 Code deux adresses

Les codes deux adresses autorisent les opérations du type  $x \leftarrow x \text{ op } y$  (on écrit  $x \leftarrow \text{op } y$ ). Ils détruisent donc un de leurs opérandes. Encore une fois, ils sont surtout utiles pour les machines qui ont un jeu d'instructions deux adresses (PDP-11)

Voici trois représentations linéaires de  $x - 2 \times y$ , en code une adresse, deux adresses ou trois adresses.

<i>push</i>	2	$t_1 \leftarrow 2$	$t_1 \leftarrow 2$
<i>push</i>	<i>y</i>	$t_2 \leftarrow y$	$t_2 \leftarrow y$
<i>multiply</i>		$t_1 \leftarrow \times t_2$	$t_3 \leftarrow t_1 \times t_2$
<i>push</i>	<i>x</i>	$t_3 \leftarrow x$	$t_4 \leftarrow x$
<i>subtract</i>		$t_3 \leftarrow -t_1$	$t_5 \leftarrow t_4 - t_3$

### 5.2.3 Code trois adresses

Les codes trois adresses autorisent les instructions du type :  $x \leftarrow y \text{ op } z$  avec au plus trois noms (en pratique il autorise des instructions avec moins de trois opérandes aussi). Ci dessus, on montre un exemple de notation infixe. L'absence de destruction des opérateurs donne au compilateur plus de souplesse pour gérer les noms. Il y a plusieurs manières d'implémenter le code trois adresses. La manière la plus naturelle et la plus souple est avec des quadruplets (cible, opération, deux opérandes, c.f. représentation ci-dessous à gauche), mais on peut aussi utiliser des triplets (l'index de l'opération nomme son registre cible), le problème est que la réorganisation du code est difficile car l'espace des noms est directement dépendant du placement de l'instruction. On peut alors faire deux tables (triplets indirects). Cette dernière solution à deux avantages sur le quadruplet : on peut réordonner très facilement les instructions et d'autre part, si une instruction est utilisée plusieurs fois elle peut n'être stockée qu'une fois. Le problème de ce stockage est qu'il fait des accès mémoire en deux endroits différents. Voici les trois propositions de stockage du

code trois adresses pour la même expression  $x - 2 \times y$ .

Target	Op	Arg <sub>1</sub>	Arg <sub>2</sub>
t <sub>1</sub>		2	
t <sub>2</sub>		y	
t <sub>3</sub>	×	(1)	(2)
t <sub>4</sub>		x	
t <sub>5</sub>	-	(4)	(3)

Op	Arg <sub>1</sub>	Arg <sub>2</sub>
	2	
	y	
×	(1)	(2)
	x	
-	(4)	(3)

Instr	Op	Arg <sub>1</sub>	Arg <sub>2</sub>
(1)		2	
(2)		y	
(3)	×	(1)	(2)
(4)		x	
(5)	-	(4)	(3)

Encore une fois, ne nombreux processeur récents ont des instructions à plus de trois adresses (le FMA des PowerPC et Itanium en a 4, la plupart des instructions ARM aussi). Cela n'apporterait rien de plus d'avoir du code 4 adresse dans l'IR : le code 3 adresses n'est pas utile parce qu'il colle au jeu d'instruction, mais parce qu'il ne détruit aucun opérande. C'est cette propriété des jeux d'instructions récents qui est importante, et passer à 4 adresses n'apporterait rien de plus.

### 5.3 Static Single Assignment

Le code assembleur en assignation unique (*static single assignment*, SSA) est de plus en plus utilisé car il ajoute de l'information à la fois sur le flot de contrôle et sur le flot des données. Un programme est en forme SSA quand

1. chaque nom de variable est défini une seule fois.
2. chaque utilisation de variable fait référence à une seule définition.

Attention, l'exécution pourra passer plusieurs fois sur la définition d'une variable dans une boucle : il s'agit d'assignation unique *statique* (dans l'IR), mais l'assignation *dynamique* pourra être multiple (sans quoi la consommation mémoire serait difficile à gérer...).

Considérons l'exemple suivant :

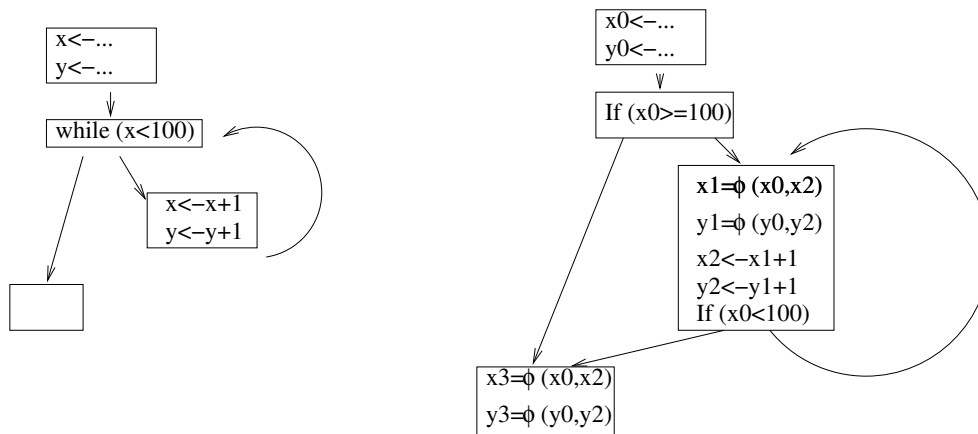
```
x ← ...
y ← ...
while (x < 100)
  x ← x + 1
  y ← y + x
end
```

Le passage en SSA va donner :

```
x0 ← ...
y0 ← ...
if (x0 ≥ 100) goto next
loop :
  x1 ← φ(x0, x2)
  y1 ← φ(y0, y2)
  x2 ← x1 + 1
  y2 ← y1 + x2
  if (x2 < 100) goto loop
next : x3 ← φ(x0, x2)
      y3 ← φ(y0, y2)
```

Dans  $y_1 \leftarrow \phi(y_0, y_2)$ , la  $\phi$ -fonctions ( $\phi$  comme fusion sans doute) signifie " $y_1$  a une valeur qui peut provenir, selon l'exécution du programme, soit de  $y_0$  par la flèche de gauche entrant dans ce bloc, soit de  $y_2$  par la flèche de droite".

En pratique on utilisera un CFG en SSA :



Pour transformer un code arbitraire en SSA, on le transforme d'abord en un graphe de flot de contrôle avec des blocs de base comme précédemment. Ensuite, on renomme les variables pour créer l'assignation unique. Lorsqu'un bloc a plusieurs prédécesseurs (plusieurs arêtes entrantes), certaines variable utilisée dans ce bloc peuvent avoir des définitions provenant de ces différentes arêtes. On aura déjà renommé ces définitions avec des noms différents. Il reste, en entrée des blocs de base, à insérer des  $\phi$  - fonctions qui décrivent les différentes options et les unifient dans une nouvelle variable.

On voit que les  $\phi$  - fonctions sera placées en entrée des blocs de base ayant plusieurs arêtes entrantes, et auront autant d'argument qu'il y a d'arête entrante au bloc de base.

On utilise fréquemment la relation de dominance entre instructions :  $a$  domine  $b$  signifie : "tout chemin arrivant à  $b$  passe par  $a$ ". La définition formelle utilisée pour la SSA est la suivante :

1. chaque nom de variable est défini une seule fois.
2. Si  $x$  est utilisé dans une instruction  $s$  et que  $s$  n'est pas un fonction  $\phi$ , alors la définition de  $x$  domine  $s$ .
3. Si  $x$  est le  $i$ ème argument d'une fonction  $\phi$  dans le bloc  $n$ , alors la définition de  $x$  domine la dernière instruction du  $i$ ème prédécesseur de  $n$ .

Cette notion de dominance, qui est présente dans la représentation SSA, sera très utile pour des tas d'optimisations.

Bien sûr, les fonctions  $\phi$  ne sont jamais implémentées dans le code généré, elle peuvent être supprimées en introduisant des instructions de copies (placées sur de nouveaux blocs sur les arêtes entrantes).

En revanche, on doit les manipuler dans la représentation intermédiaire. Elle présentent deux singularités par rapport à des instructions habituelles : lorsqu'elles sont exécutées, certains de leurs arguments peuvent ne pas avoir été définis et d'autre part elles peuvent avoir un nombre quelconque d'arguments (car un bloc de base peut avoir un nombre quelconque d'arêtes entrantes, par exemple dans le cas du *case*).

## 5.4 Modèle mémoire dans les IR

Jadis, il y a eu des processeurs dont le jeu d'instruction permettait du calcul mémoire à mémoire. Par exemple, une instruction pouvait ajouter à un registre le contenu d'une case mémoire. Du fait de l'écart croissant entre les performances de la mémoires et celles du processeur, il est de plus en plus nécessaire de limiter au maximum les accès à la mémoire. C'est une des raisons pour laquelle les processeurs RISC calculent sur les registres uniquement. Les seules opérations qu'ils peuvent faire sur la mémoire consistent à charger le contenu d'une case mémoire dans un registre (*load*), et à stocker le contenu d'un registre dans une case mémoire (*store*). On parle d'architecture *load-store*.

Cependant, la plupart des langages de programmation ont une sémantique pour les variables qui est définie en terme de cases mémoires. C'est évident pour les tableaux et parfois pour les structures, mais c'est plus général que cela. Par exemple, en C, le simple fait qu'on puisse par

l'opération & obtenir l'adresse d'une variable de (presque) n'importe quel type illustre bien que la sémantique est orientée mémoire.

Une partie de la difficulté à définir une bonne IR est de permettre ce passage de la sémantique orientée mémoire du langage source à la sémantique orientée registres du langage destination. Une bonne solution est la suivante :

- Toutes les variables *du programme source* sont traitées comme des cases mémoire, et remplacées par des pointeurs sur cette case lors de la génération de code. Par conséquent, un accès à une variable en partie droite d'une assignation se traduira dans l'IR par une lecture mémoire, et une variable en partie gauche se traduira par une écriture mémoire.

Par exemple, le code suivant :

```
i = i+1;
if (i==10) { ... }
```

se traduira par le code ILOC suivant :

```
loadI @i    => r1    //adresse de i dans r1, après on se souvient qu'elle est là
load  (r1)  => r2    //valeur de i dans r2
loadI 1     => r3    //valeur 1 dans r3
add   r2,r3 => r4
store r4    => (r1) //fin de la première ligne

load  (r5)  => r1    //valeur de i dans r7 (certes on l'avait déjà dans r4)
...
```

Autrement dit, la génération initiale de code, qui produit le code 3 adresses de départ, produit un code non optimisé mais correspondant à la sémantique du langage.

- Toutes les autres variables de notre code 3 adresses (qui ne sont donc pas des variables du programme source) seront moralement des registres (ils sont appelés  $r_*$  dans le code précédent). Comme on en crée un grand nombre, la génération de code à partir de l'IR aura une phase obligatoire d'*allocation de registre*. Si le processeur cible ne dispose pas d'assez de registres, certains de ces registres seront stockés en mémoire (et seront donc plus coûteux que les autres à l'exécution, donc on les choisira soigneusement).
- Les accès mémoire inutiles dans le code ainsi généré (voir l'exemple) pourront être supprimés par la suite lors d'une phase facultative d'optimisation. Cette phase pourra aussi chercher à conserver dans des registres les variables du programme si cela ne change pas sa sémantique. C'est parfois plus compliqué qu'il n'y paraît. Par exemple, en C, il ne suffit pas de vérifier qu'il n'y a pas de  $\&x$  dans le programme pour en déduire qu'on peut conserver  $x$  dans un registre. Grâce à une arithmétique de pointeur très ouverte, à peu près n'importe quel pointeur du programme<sup>2</sup> peut se retrouver pointer sur  $x$  ! En pratique, sur le domaine de définition (*scope*) d'une variable, il est tout de même courant qu'il n'y ait pas d'arithmétique de pointeur à risque. Au passage, cela montre que faire des gourouteries sur des pointeurs sur `void` ne va pas toujours produire du code plus efficace : cela pourra aussi inhiber des optimisations que le compilateur aurait pu faire.

Une remarque sur cette approche et la SSA : en SSA, une fois effectués les chargements des valeurs initiales dans des registres, toutes les variables sont des registres. Les écritures en mémoires utiles sont déterminées lors de la génération du code final à partir de la SSA. Donc la traduction du `while` qui était notre exemple SSA correspond bien à ce qu'on obtiendra en pratique. Et si il y a un risque que quelqu'un d'autre écrive à l'adresse de  $x$ , alors au lieu de  $\phi$  on aura des relectures de la mémoire. Ce risque de corruption mémoire est évalué lors de la génération initiale de code, en regardant juste les pointeurs en jeu.

Pour illustrer la génération de code avant optimisation, voici encore le codage dans un IR

<sup>2</sup>Toutefois il y a des cas dans lesquels la norme dit que le comportement est indéfini, et on n'a pas à s'inquiéter de ces cas. Par exemple, un programme qui accède à `A[i]` pour `i` en dehors du domaine défini a un comportement indéfini, donc le compilateur ne doit pas faire la supposition qu'une écriture dans un tableau pourra écrire dans toute variable : c'est certes vrai, mais dans ce cas le compilateur est déchargé de toute responsabilité sur le contenu de  $x$  et le reste de la bonne marche de l'univers.

linéaire trois adresses du type ILOC d'un simple appel de tableau  $A[i, j]$  :

```

loadI 1      ⇒ r1
sub   rj, r1 ⇒ r2    //r2 ← j - 1
loadI 10     ⇒ r3    // le tableau A est rangé par ligne, il des ligne de taille 10
mult  r2, r3 ⇒ r4
sub   ri, r1 ⇒ r5
add   r4, r5 ⇒ r6    //r6 ← 10 × (j - 1) + i - 1
loadI @A     ⇒ r7
add   r7, r6 ⇒ r8
load  (r8)   ⇒ rAij  //rAij ← @A + 10 × (j - 1) + i - 1

```

## 5.5 La table des symboles

Le compilateur va rencontrer un nombre important de symboles, correspondant à beaucoup d'objets différents (nom de variables, constantes, procédures, labels, types, temporaires générés par le compilateur...). Pour chaque type d'objet le compilateur stocke des informations différentes. On peut choisir de conserver ces informations soit directement sur l'IR (stocker le type d'une variable sur le noeud de l'AST correspondant à sa déclaration par exemple) ou créer un répertoire central pour toutes ces informations et proposer un mécanisme efficace pour y accéder. On appelle ce répertoire la *table des symboles* (rien n'empêche qu'il y ait plusieurs tables). L'implémentation de la table des symboles nécessite une attention particulière car elle est très sollicitée lors de la compilation.

**Remarque importante** La table des symboles est utilisée par le compilateur *pendant la compilation*, comme l'IR. Les mécanismes d'accès aux variables à l'*exécution* que le compilateur devra mettre en place sont décrits dans le chapitre suivant.

Le mécanisme le plus efficace pour ranger des objets qui ne sont pas pré-indexés est la table de hachage (*hash table*). Ces structures ont une complexité en moyenne de  $O(1)$  pour insérer et rechercher un objet et peuvent être étendues relativement efficacement. On utilise pour cela une fonction de hachage  $h$  qui envoie les noms sur des petits entiers qui servent d'index dans la table. Lorsque la fonction de hachage n'est pas *parfaite*, il peut y avoir des collisions ; deux noms donnant le même index. On utilisera les primitives :

- *recherche(nom)* qui retourne les informations stockées dans la table pour le nom *nom* (adresse  $h(\text{nom})$ )
- *insérer(nom, info)* pour insérer l'information *info* à l'adresse  $h(\text{nom})$ . Cela peut éventuellement provoquer l'expansion de la table.

Dans le cas d'un langage n'ayant qu'une seule portée syntaxique, cette table peut servir directement à vérifier des informations de typage telles que la déclaration d'une variable que l'on utilise, la cohérence du type etc. Malheureusement, la plupart des langages proposent différentes portées, généralement imbriquées, pour les variables. Par exemple en C, une variable peut être globale (tous les noms correspondant à cette variable réfèrent la même case mémoire), locale à un fichier (avec le mot clé *static*), locale à une procédure ou locale à un bloc.

Par exemple :

```

static int w;      /* niveau 0 */
int x;

void example(a, b)
{
    int a, b;      /* niveau 1 */
    {
        int c;
        {
            int b, z;    /* niveau 2a */
        }
        {
            int a, x;    /* niveau 2b */
        }
    }
}

```

```

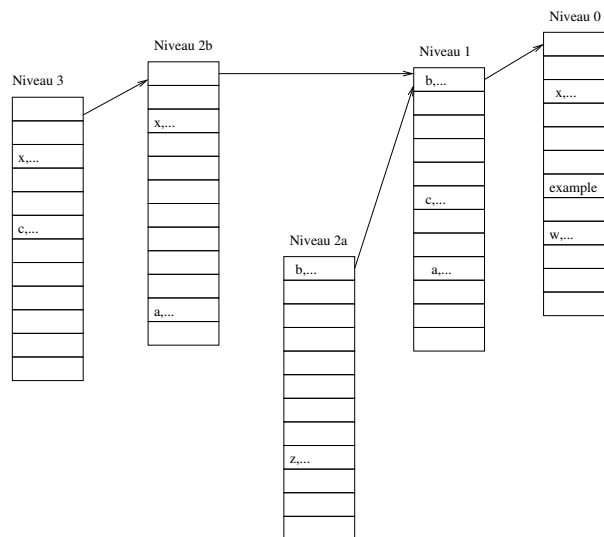
...
{
  int c, x;          /* niveau 3 */
  b = a + b + c + x;
}
}
}

```

Pour compiler correctement cela, le compilateur a besoin d'une table dont la structure permet de résoudre une référence à la définition la plus récente lexicalement. Au moment de la compilation il doit émettre le code effectuant la bonne référence.

On va donc convertir une référence à un nom de variable (par exemple  $x$ ) en un couple  $\langle \text{niveau}, \text{index} \rangle$  ou *niveau* est le niveau lexical dans lequel la définition de la variable référencée a lieu et *index* est un entier identifiant la variable dans la portée considérée. On appelle ce couple la coordonnée de distance statique (*static distance coordinate*).

La construction et la manipulation de la table des symboles va donc être un peu différente. Chaque fois que le parseur entre dans une nouvelle portée lexicale, il crée une nouvelle table des symboles pour cette portée. Cette table pointe sur *sa mère* c'est à dire la table des symboles de la portée immédiatement englobante. La procédure *Inserer* est la même, elle n'affecte que la table des symboles courante. La procédure *recherche* commence la recherche dans la table courante puis remonte d'un niveau si elle ne trouve pas (et ainsi de suite). La table des symboles pour l'exemple considéré ressemblera donc à cela :



Les noms générés par le compilateur peuvent être stockés dans la même table des symboles.

Tout ceci ne concerne que la structure utilisée en interne par le compilateur. On verra bientôt comment ces variables sont stockées et gérées au niveau du code produit.

## 6 Gestion de la mémoire dans les langages impératifs

À partir de cette section, les langages impératifs et les langages fonctionnels divergent. On va traiter surtout les langages impératifs dans un premier temps.

Pour la plupart de ces langages, la procédure est l'unité de base. Ce sera donc celle du compilateur. Cela se traduit par la notion de *compilation séparée* des procédures. C'était indispensables aux temps héroïques de l'informatique. De nos jours, le moindre PC peut en théorie gérer dans sa mémoire centrale des programmes énormes, mais on continue à faire de la compilation séparée qui a de nombreux avantages :

- gain de temps lors du développement : on ne recompile que ce qui est nécessaire,
- dualement, possibilité de passer plus de temps à faire des optimisations poussées sur des morceaux de codes assez petits,
- possibilité de distribuer des bibliothèques pré-compilées, sans distribuer le code source
- Possibilité d'édition de lien dynamique.

Tous ces avantages viennent au prix d'une étape d'*édition de lien*, qui assemble le code compilé des procédures pour en faire un programme complet. L'édition de lien est un contrat entre le compilateur, le système d'exploitation et la machine cible qui répartit les responsabilités pour nommer, pour allouer des ressources, pour l'adressabilité et pour la protection. L'édition de lien autorise à générer du code pour une procédure indépendamment de la manière avec laquelle on l'appelle.

Nous verrons (peut-être, dans longtemps) qu'il est possible de faire toutefois des optimisations *interprocédurales*.

Du point de vue du langage, une procédure propose trois abstractions importantes :

- Abstraction du contrôle. Le mécanisme standard de passage des paramètres et du résultat entre la procédure appelée et le programme appelant permet d'appeler une procédure sans connaître comment elle est implémentée.
- L'espace des noms. Chaque procédure crée un nouvel espace des noms protégé pour ses variables locales. C'est une notion importante d'ordre syntaxique, et donc liée au code source (définition de la portée d'une variable, référence aux paramètres effectifs indépendamment de leur nom à l'extérieur, etc) mais cela se traduit par des mécanismes spécifiques d'accès à ces variables locales : il faut leur allouer de l'espace à l'appel de la procédure, et libérer cet espace à la fin de l'exécution de la procédure, tout cela de manière automatique et efficace.
- Interface externe. Les procédures définissent les interfaces entre les différentes parties d'un gros système logiciel. On peut par exemple, appeler des bibliothèques graphiques ou de calculs scientifiques. En général le système d'exploitation utilise la même interface pour appeler une certaine application : appel d'une procédure spéciale, *main*.

La compilation d'une procédure doit faire le lien entre la vision haut niveau (espace des noms hiérarchique, code modulaire) et l'exécution matérielle qui voit la mémoire comme un simple tableau linéaire et l'exécution comme une simple incrémentation du *program counter* (PC). Ce cours s'applique essentiellement à la classe des langages à la *Algol* qui a introduit la programmation structurée.

La compilation séparée est encore plus intéressante si l'on peut appeler depuis un programme en C une bibliothèque écrite en Fortran, et réciproquement. Pour que ce soit possible, il faut que l'interface des procédures au niveau du code objet soit compatible avec tous les langages. Cette interface doit donc offrir l'union de tous les langages cibles envisagés, et pourra donc paraître inutilement compliquée lorsqu'on regarde le code compilé pour un langage simple comme C. Deux exemples :

- Pascal a des procédures imbriquées, pas C (sauf que `gcc` les accepte car c'est une extension GNU à C).
- Les langages objets nécessitent des mécanismes spécifiques, que nous verrons aussi.

### 6.1 Vue d'ensemble

Le mécanisme de transfert de contrôle entre les procédures est le suivant : lorsqu'on appelle une procédure, le contrôle est donné à la procédure appelée ; lorsque cette procédure appelée se termine, le contrôle est redonné à la procédure appelante, juste après l'appel. Deux appels à une

même procédure créent donc deux *instances* (ou invocations) indépendantes. On définit le *graphe d'appel* (arbre des appels possibles) et l'*historique de l'exécution* qui est un parcours de cet arbre.

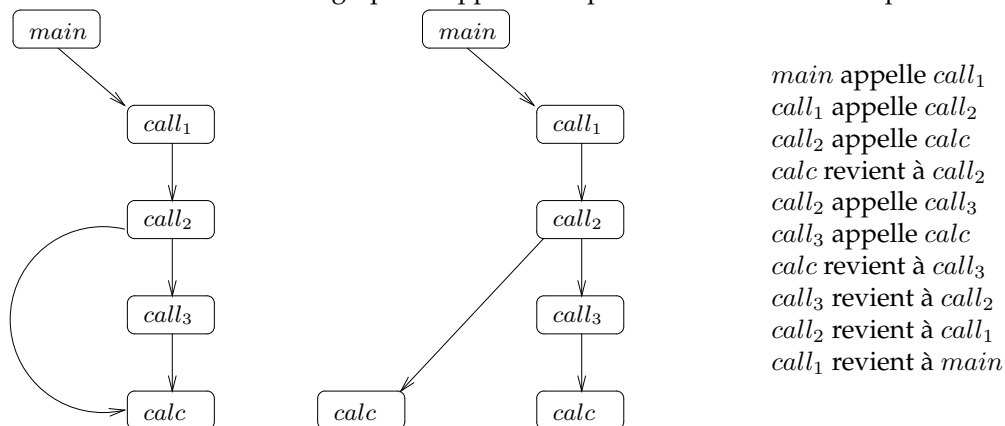
Considérons le programme Pascal suivant :

```

program Main(input, output);
  var x,y,...
  procedure calc;
  begin { calc}
  ...
  end;
  procedure call1;
  var y...
  procedure call2
  var z : ...
  procedure call3;
  var y...
  begin { call3}
  x :=...
  calc;
  end;
  begin { call2}
  z :=1;
  calc;
  call3;
  end;
  begin { call1}
  call2;
  ...
  end;
begin { main }
...
call1;
end;

```

La figure suivante illustre la différence entre le graphe d'appel (*call graph*) et l'arbre d'appel (qui est la version déroulée du graphe d'appel correspondant à une exécution particulière).



Ce mode d'appel et de retour s'exécute bien avec une pile : Lorsque *call<sub>1</sub>* appelle *call<sub>2</sub>*, il pousse l'adresse de retour sur la pile. Lorsque *call<sub>2</sub>* finit, il dépile l'adresse de retour et branche à cette adresse. Ce mécanisme fonctionne aussi avec les appels récursifs. Certains ordinateurs (dont le x86) ont cablé ces accès à la pile dans les instructions d'appel et de retour de procédure.

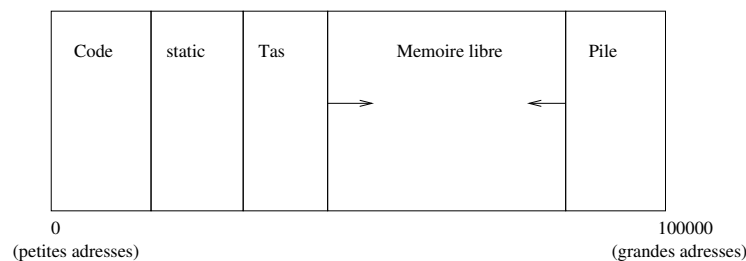
De plus, il devient alors naturel de stocker sur la même pile toutes les variables locales à la procédure : cela fournira un mécanisme implicite d'allocation et de déallocation de cette mémoire



locale lors des appels de procédures (éventuellement récursifs). En pratique ce n'est pas plus coûteux de réserver la place pour 36 variables que d'empiler une valeur : il suffit, lors d'un appel de procédure, d'augmenter le pointeur de pile de la taille mémoire correspondant à ces variables locales. Ensuite, le code s'interdit de toucher au pointeur de pile tant que l'exécution reste dans cette procédure, si bien que toutes les adresses des variables locales sont définies par un *offset* (décalage, addition d'une constante) par rapport au pointeur de pile. Cet offset est défini par le compilateur, rangé dans la table des symboles, et utilisé lors de la génération de code pour produire l'adresse de chaque variable.

La suite détaille ce mécanisme. Il est à noter que tous les langages permettent aussi d'allouer de la mémoire dynamiquement (par exemple pour les variables dont la taille n'est pas connue à la compilation). Cette mémoire est allouée sur le *tas*. En Java, par exemple, le tout-venant des variables (entiers, flottants, pointeurs sur les objets) est en général alloué dans la pile, mais les objets sont eux alloués sur le *tas*.

D'un point de vue "vision logique de la mémoire par le programme", la mémoire linéaire est généralement organisée de la façon suivante : la pile sert à l'allocation des appels de procédures (elle grandit par exemple vers les petites adresses), le *tas* sert à allouer la mémoire dynamiquement (il grandit alors vers les grandes adresses). La gestion mémoire globale (gestion du *tas*, ramasse-miette, et mémoire virtuelle) sont détaillés en annexe.



Remarque : Certain langages autorisent une procédure à retourner une autre procédure et son environnement à l'exécution (clôture). La procédure retournée s'exécute alors dans l'environnement retourné. Une simple pile n'est pas suffisante pour cela.

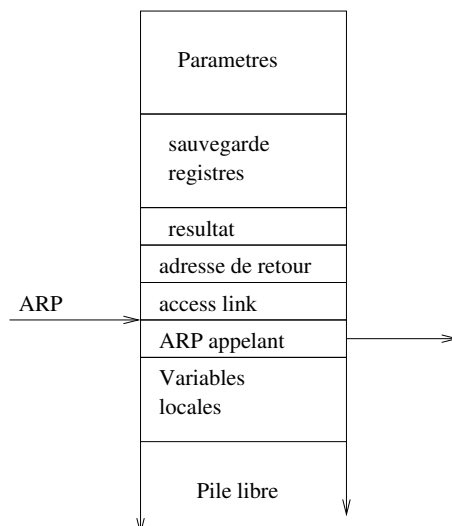
## 6.2 Enregistrement d'activation (AR)

L'espace mémoire mis en place en haut de la pile lors de l'appel d'une procédure est appelé *Enregistrement d'activation* (*activation record* : AR). Sauf cas exceptionnel, cet espace est désalloué lorsque la procédure se termine.

L'enregistrement d'activation d'une procédure  $q$  est accédé par le pointeur d'enregistrement d'activation (*activation record pointer*, ARP souvent appelé *frame pointer* : FP). L'enregistrement d'activation comporte la place pour

- les paramètres effectifs,
- les variables locales à la procédure,
- la sauvegarde de l'environnement d'exécution de la procédure appelante (essentiellement les registres)
- l'adresse de retour de la procédure,
- un lien d'accès permettant d'accéder à d'autres données (*access link*),
- et un pointeur sur l'ARP de la procédure appelante.

Tout ceci va être détaillé. Vous êtes invité à visualiser tout cela en terme de votre langage favori, mais il pourra alors vous sembler que cette information est redondante. Il faut se souvenir que l'AR est défini pour accomoder tous les langages que l'on désire faire communiquer ensemble. Par exemple, l'*access link* est inutile si on ne fait que du C.



La plupart des compilateurs dédient un registre au stockage de l'ARP. Ici, on supposera que ou  $r_{arp}$  contient en permanence l'ARP courant. Ce qui est proche de l'adresse pointée par l'ARP est de taille fixe, le reste (généralement de part et d'autre de l'ARP peut être de taille dépendant de la procédure).

### 6.2.1 Variables locales

Pour chaque procédure, le compilateur calcule la taille des variables locales. Pour chaque variable, il référence son niveau d'imbrication, et lui attribue un offset par rapport à l'ARP. Il enregistre ces informations dans la table des symboles. La variable peut alors être accédée par l'instructions :

$$loadAI\ r_{arp},\ offset$$

Lorsque la taille de la variable ne peut pas être connue statiquement (provenant d'une entrée du programme – ce n'est jamais le cas en C, par exemple), le compilateur définit juste un pointeur sur une zone mémoire qui sera allouée sur le tas (*heap*) lorsque sa taille sera connue. Le compilateur doit alors générer un code d'accès légèrement différent, l'adresse de la variable est récupérée avec *loadA0*, puis un deuxième load permet d'accéder à la variable. Bien sûr, si celle-ci est accédée souvent, le compilateur peut choisir de conserver son adresse dans un registre.

Le compilateur doit aussi se charger de l'initialisation éventuelle des variables. Dans le cas général il doit générer le code permettant d'initialiser la variable lors de chaque appel de procédure (c.a.d. rajouter des instructions avant l'exécution de la première instruction de la procédure).

Remarque : Certains langages permettent de définir des variables locales *statiques* (dont la durée de vie est indépendante de la procédure). Elles sont gérées comme des variables globales.

### 6.2.2 Sauvegarde des registres

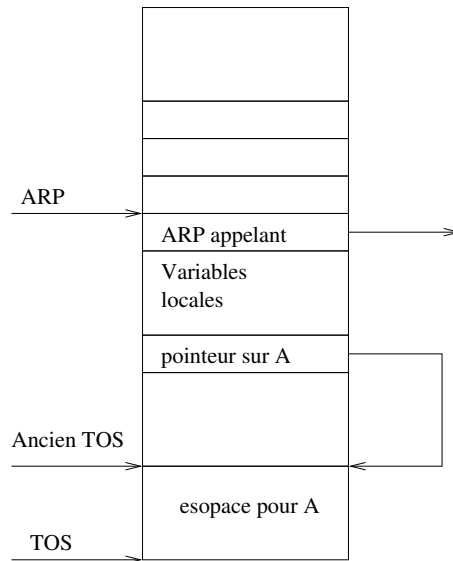
Lorsque  $p$  appelle  $q$ , soit  $p$  soit  $q$  doit se charger de sauvegarder les registres (ou au moins une partie d'entre eux qui représentent le contexte d'exécution de la procédure  $p$ ) et de les restaurer en fin de procédure.

### 6.2.3 Allocation de l'enregistrement d'activation

En général la procédure appelante ne peut pas allouer complètement l'enregistrement d'activation de la procédure appelée : il lui manque des données comme la taille des variables locales. Ce qu'elle a alloué, c'est ce qui dépend d'elle : elle a empilé les paramètres effectifs, par exemple.

Pour le reste, lorsque la durée de vie de l'AR est limitée à celle de la procédure (cas général), le compilateur peut l'allouer aussi sur la pile. Le surcoût de l'allocation et de la désallocation est

alors faible : c'est une addition sur le pointeur de pile. On peut aussi implémenter des données de tailles variables sur la pile, en modifiant dynamiquement la fin de l'AR. Ceci explique que l'ARP est distinct du pointeur de pile.



Par exemple, voici le code que gcc produit typiquement en entrée d'une procédure (`esp` est le pointeur de pile, `ebp` est l'ARP, et la pile descend dans la mémoire comme sur la figure précédente) :

```
.main :
pushl %ebp    (empiler le lien vers l'ARP de l'appelant)
movl %esp, %ebp (l'ARP de cette procédure sera le pointeur de pile à cet instant)
subl $24, %esp (réservé sur la pile l'espace pour les variables locales)
```

Remarque : ce code n'est pas cohérent avec les figures précédentes... Elles doivent décrire ce qui se passe sur un autre système d'exploitation, en effet tout ceci est défini par l'OS.

Cas particuliers :

- Dans certains cas, la procédure doit survivre plus longtemps que son appel (par exemple si la procédure retourne une clôture qui contient des références aux variables locales à la procédure). Dans ce cas, on doit allouer l'AR sur le tas. On désalloue une zone lorsque plus aucun pointeur ne pointe dessus (ramasse miette).
- Dans le cas où une procédure  $q$  n'appelle pas d'autre procédure,  $q$  est appelée une feuille. Dans ce cas, l'AR peut être alloué statiquement et non à l'exécution (on sait qu'un seul  $q$  sera actif à un instant donné). Cela gagne le temps de l'allocation à l'exécution (en général c'est clopinettes).
- Dans certains cas, le compilateur peut repérer que certaines procédures sont toujours appelées dans le même ordre (par exemple,  $call_1$ ,  $call_2$  et  $call_3$ ). Il peut alors décider de les allouer en même temps (avantageux surtout dans le cas d'une allocation sur le tas).

### 6.3 Communication des valeurs entre procédures

Il y a deux modes principaux de passage des paramètres entre les procédures : passage par valeur et passage par référence. Considérons l'exemple C suivant :

```
int proc(x, y)
{
    int x, y;
    x = 2 * x;
    y = x + y;
}

c = proc(2, 3);
a = 2;
b = 3;
c = proc(a, b);
a = 2
```

```

    return y;
}
    b = 3;
    c = proc(a, a);

```

Dans ce cas, il y a passage de paramètres par valeurs : l'appelant copie les valeurs du paramètre effectif dans l'espace réservé pour cela dans l'AR. Le paramètre formel a donc son propre espace de stockage, seul son nom réfère à cette valeur qui est déterminée en évaluant le paramètre effectif au moment de l'appel. Voici le résultat de l'exécution, qui est généralement assez intuitif (le passage par valeur a les faveurs des langages modernes pour cette raison) :

<i>appel par valeur</i>	<i>a in</i>	<i>a out</i>	<i>b in</i>	<i>b out</i>	<i>resultat</i>
<i>proc(2, 3)</i>	—	—	—	—	7
<i>proc(a, b)</i>	2	2	3	3	7
<i>proc(a, a)</i>	2	2	3	3	6

Lors d'un appel par référence, l'appelant stocke dans l'AR de l'appelé un pointeur vers l'endroit où se trouve le paramètre (si c'est une variable, c'est l'adresse de la variable, si c'est une expression, il stocke le résultat de l'expression dans son propre AR et indique cette adresse). Un appel par référence implique donc un niveau de plus d'indirection, cela implique deux comportements fondamentalement différents : (1) la redéfinition du paramètre formel est transmise au paramètre effectif et (2) un paramètre formel peut être lié à une autre variable de la procédure appelée (et donc modifiée par effet de bord de la modification de la variable locale), ce qui est un comportement très contre-intuitif. Voici un exemple en PL/I qui utilise le passage par références :

```

procEDURE proc(x, y)
  RETURNS FIXED BINARY
  DECLARE x, y FIXED BINARY;
  BEGIN
    x = 2 * x;
    y = x + y;
    RETURN y;
  END
    c = proc(2, 3)
    a = 2;
    b = 3;
    c = proc(a, b)
    a = 2
    b = 3;
    c = proc(a, a);

```

et le résultat de l'exécution :

<i>appel par valeur</i>	<i>a in</i>	<i>a out</i>	<i>b in</i>	<i>b out</i>	<i>resultat</i>
<i>proc(2, 3)</i>	—	—	—	—	7
<i>proc(a, b)</i>	2	4	3	7	7
<i>proc(a, a)</i>	2	8	3	8	8

Il existe d'autres méthodes exotiques pour passer les paramètres ; par exemple, le passage par nom (Algol) : Une référence au paramètre formel se comporte exactement comme si le paramètre effectif avait été mis à sa place. Le passage par valeur/résultat (certains compilateurs fortran) exécute le code de la procédure comme dans un passage par valeur mais recopie les valeurs des paramètres formels dans les paramètres effectifs à la fin (sauf si ces paramètres sont des expressions).

La valeur retournée par la procédure doit être en général stockée en dehors de l'AR car elle est utilisée alors que la procédure a terminé. L'AR inclut une place pour le résultat de la procédure. La procédure appelante doit alors elle aussi allouer de la place pour le résultat dans son propre AR et stocker un pointeur dans la case prévue pour le résultat dans l'AR de l'appelant. Mais si la taille du résultat n'est pas connue statiquement, il sera probablement alloué sur le tas et la procédure appelée stockera un pointeur sur ce résultat dans l'AR de l'appelant. Quelque soit le choix fait, il doit être compatible avec des programmes compilés par ailleurs, sous peine de provoquer des erreurs à l'exécution incompréhensibles.

## 6.4 Adressabilité

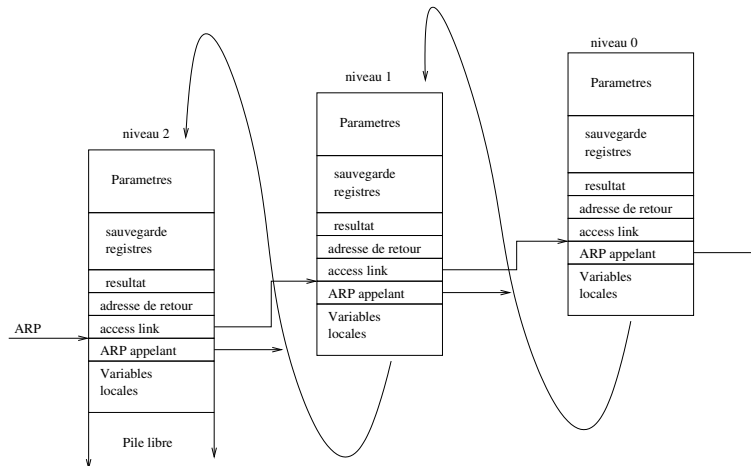
Pour la plupart des variables, le compilateur peut émettre un code qui génèrera l'adresse de base (l'adresse à partir de laquelle on compte l'offset pour retrouver la variable) puis l'offset. Le

cas le plus simple est le cas des variables locales où l'adresse de base est l'ARP. De même, l'accès aux variables globales ou statiques, est fait de la même manière, l'adresse de base étant stockée quelque part. L'endroit où est cette adresse de base est en général fixé à l'édition de lien, le compilateur introduit un nouveau label (par exemple  $\&a$  pour la variable globale  $a$ ) qui représentera l'adresse à laquelle  $a$  est stockée

Pour les variables des autres procédures, c'est un peu plus complexe ; il faut un mécanisme d'accès de ces variables à l'exécution à partir des coordonnées de distance statique définies au chapitre précédent. Plusieurs solutions sont possibles. Considérons par exemple que la procédure  $calc$  référence une variable  $a$  déclarée au niveau  $l$  dans la procédure  $call_1$ . Le compilateur a donc accès à une information du type  $\langle l, decalage \rangle$  et il connaît le niveau de  $calc$ . Le compilateur doit d'abord générer le code pour retrouver l'ARP de la procédure  $call_1$ , puis utiliser  $decalage$  pour retrouver la variable.

### 6.4.1 Lien d'accès

Dans ce schéma, le compilateur insère dans chaque AR un pointeur sur l'AR de l'ancêtre immédiat (c'est à dire dont la portée est immédiatement englobante). On appelle ce pointeur un lien d'accès (*Access link*) car il sert à accéder aux variables non locales. Le compilateur émet le code nécessaire à descendre dans les AR successifs jusqu'à l'AR recherché.



Par exemple, en supposant que le lien d'accès soit stocké avec un décalage 4 par rapport à l'ARP. Voici le type de code pour différents accès à différents niveaux (notons que quelquefois on préfère accéder aux variables en utilisant le sommet de pile (P) mais le mécanisme est similaire, avec des décalages positifs) :

$\langle 2, -24 \rangle$	$loadAI$	$r_{arp}, -24$	$\Rightarrow$	$r_2$
$\langle 1, -12 \rangle$	$loadAI$	$r_{arp}, 4$	$\Rightarrow$	$r_1$
	$loadAI$	$r_1, -12$	$\Rightarrow$	$r_2$
$\langle 0, -16 \rangle$	$loadAI$	$r_{arp}, 4$	$\Rightarrow$	$r_1$
	$loadAI$	$r_1, 4$	$\Rightarrow$	$r_1$
	$loadAI$	$r_1, -16$	$\Rightarrow$	$r_2$

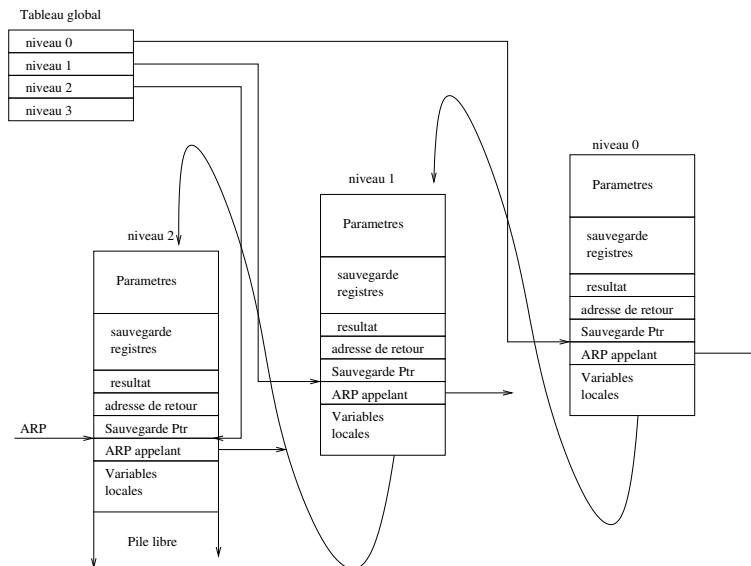
Souvent la procédure appelée est celle de niveau lexical juste englobant, le lien d'accès est alors simplement l'ARP de l'appelant mais pas toujours (voir par exemple l'appel à  $calc$  dans l'exemple plus haut). Dans ce cas, il faut maintenir le lien d'accès dans l'ARP.

Lors de la construction de l'AR, le lien d'accès est calculé en remontant par le lien d'accès de l'appelant jusqu'au niveau père commun à l'appelant et à l'appelé (qui est le niveau sur lequel il faut pointer avec le lien d'accès de l'appelé).

### 6.4.2 Accès global

On peut aussi choisir d'allouer un tableau global (*global display*) pour accéder les ARP des procédures. Chaque accès à une variable non locale devient une référence indirecte. Comme des

procédures de même niveau peuvent s'appeler, il faut un mécanisme de sauvegarde du pointeur sur chaque niveau. À l'appel d'une procédure de niveau  $l$ , on stocke le pointeur sur le niveau  $l$  dans l'AR de la procédure appelée et on le remplace (dans le tableau global) par l'AR de la procédure appelée. Lorsque l'on sort de la procédure, on restaure le pointeur de niveau  $l$  dans le tableau. L'accès aux différents niveaux présente des différences de complexité moins marquées, mais il faut deux accès à la mémoire (un à l'appel, un au retour) alors que le mécanisme du lien d'accès ne coûte rien au retour. En pratique, le choix entre les deux dépend du rapport entre le nombre de références non locales et le nombre d'appels de procédure. Dans le cas où les variables peuvent survivre aux procédures les ayant créées, l'accès global ne fonctionne pas.



Voici le type de code pour différents accès à différents niveaux :

$\langle 2, -24 \rangle$	$loadAI \ r_{arp}, -24 \Rightarrow r_2$
$\langle 1, -12 \rangle$	$loadI \ \_disp \Rightarrow r_1$
	$loadAI \ r_1, 4 \Rightarrow r_1$
	$loadAI \ r_1, -12 \Rightarrow r_2$
$\langle 0, -16 \rangle$	$loadI \ \_disp \Rightarrow r_1$
	$loadAI \ r_1, -16 \Rightarrow r_2$

## 6.5 Résumé du code à générer par le compilateur

Si une procédure  $p$  appelle une procédure  $q$ , leurs codes auront la forme suivante : chaque procédure contient un prologue et un épilogue, chaque appel contient un pré-appel et un post-appel.

- **Pre-appel** Le pré-appel met en place l'AR de la procédure appelée (allocation de l'espace nécessaire et remplissage des emplacements mémoire). Cela inclut l'évaluation des paramètres effectifs, leur stockage sur la pile, l'empilement de l'adresse de retour, l'ARP de la procédure appelante, et éventuellement la réservation de l'espace pour une valeur de retour. La procédure appelante ne peut pas connaître la place nécessaire pour les variables locales de la procédure appelée.
- **Post-appel** Il y a libération des espaces alloués lors du pré-appel.
- **Prologue** Le prologue crée l'espace pour les variables locales (avec initialisation éventuelle). Si la procédure contient des références à des variables statiques, le prologue doit préparer l'adressabilité en chargeant le label approprié dans un registre. Éventuellement, le prologue met à jour le tableau d'accès global (*display*)
- **Épilogue** L'épilogue libère l'espace alloué pour les variables locales, il restaure l'ARP de l'appelant et saute à l'adresse de retour. Si la procédure retourne un résultat, la valeur est transférée là où elle doit être utilisée.

De plus, les registres doivent être sauvegardés, soit lors du pre-appel de l'appelant (caller-save) soit lors du prologue de l'appelé (callee save). Voici un exemple de division des tâches (sans mentionner les liens d'accès ou le tableau d'accès global) :

	appelé	appelant
appel	pre-appel	prologue
	alloue les bases de l'AR évalue et stocke les paramètres stocke l'adresse de retour et l'ARP sauve les registres (caller-save) saute à l'adresse appelée	préserve les registres (callee save) étend l'AR pour les données locales trouve l'espace des données statiques initialise les variables locales exécute le code
retour	post-appel	épilogue
	libère les bases de l'AR restaure les registres (callee save) transmet le résultat et restaure les paramètres par référence	restaure les registres (callee save) libère les données locales restaure l'ARP de l'appelant  saute à l'adresse de retour

En général, mettre des fonctionnalités dans le prologue ou l'épilogue produit du code plus compact (le pre-appel et post-appel sont écrits pour chaque appel de procédure).

## 7 Désucreage des langages impératifs

Le *désucreage* consiste à enlever le sucre syntaxique qui fait les langages civilisés pour se ramener à du (pseudo)-code représentable dans la représentation intermédiaire, typiquement des blocs de base et des branchements conditionnels entre les blocs de base. Le chapitre précédent a présenté le désucreage des procédures et de leurs appels. Voyons maintenant le reste.

### 7.1 Stockage des valeurs

En plus des données du programme il y a de nombreuses autres valeurs qui sont stockées comme valeurs intermédiaires par le compilateur (par exemple, les tableaux pour l'accès global au chapitre précédent). On a vu qu'une donnée pouvait, suivant sa portée et son type, être stockée en différents endroits de la mémoire (on parle de classe de stockage, *storage class*). On distingue le stockage local à une procédure, le stockage statique d'une procédure (objet d'une procédure destiné à être appelé en dehors de l'exécution de la procédure), le stockage global et le stockage dynamique (tas). Ces différentes classes de stockage peuvent être implémentées de différentes manières.

Le compilateur doit aussi décider si une valeur peut résider dans un registre. Il peut dans ce cas l'allouer dans un registre virtuel, qui sera, ou pas, recopié en mémoire par la phase d'allocation de registres (*spill*). Pour être sûr qu'une valeur peut être stockée dans un registre, le compilateur doit connaître le nombre de noms distincts dans le code qui nomment cette valeur. Par exemple dans le code suivant, le compilateur doit assigner une case mémoire pour la variable *a*

```
void calc()
{
    int a,*b;
    ...
    b = &a;
    ...
}
```

Il ne peut même pas laisser la variable *a* dans un registre entre deux utilisations à moins d'être capable de prouver que le code ne peut avoir accès à *\*b* entre les deux. Une telle valeur est une valeur *ambiguë*. On peut créer des valeurs ambiguë avec les pointeurs mais aussi avec le passage de paramètres par référence. Les mécanismes pour limiter l'ambiguïté sont importants pour les performances de l'analyse statique (ex : le mots clés *restrict* en C qui limite le type de données sur lequel peut pointer un pointeur).

Il faut aussi tenir compte d'un certain nombre de règles spécifiques aux machines cibles. Il peut y avoir plusieurs classes de registres (floating point, general purpose, ...). Les registres peuvent être distribués sur différents *register file* (fichier de registres) qui sont associés en priorités à certains opérateurs.

### 7.2 Expressions booléennes et relationnelles

Comme pour les opérations arithmétiques, les concepteurs d'architectures proposent tous des opérations booléennes natives dans les processeurs. En revanche, les opérateurs relationnels (comparaison, test d'égalité) ne sont pas normalisés. Il y a deux manières de représenter les valeurs booléennes et relationnelles : la représentation par valeur (*value representation*) et le codage par position (*positional encoding*).

Dans la représentation par valeur, le compilateur assigne des valeurs aux booléens Vrai et Faux (en général des valeurs entières) et les opérations booléennes sont programmées pour cela. Le code à générer pour une expression booléenne est alors simple : pour  $b \vee c \wedge \neg d$  le code est :

$$\begin{aligned} \text{not } r_d &\Rightarrow r_1 \\ \text{and } r_c, r_1 &\Rightarrow r_2 \\ \text{or } r_b, r_2 &\Rightarrow r_3 \end{aligned}$$



Pour la comparaison, si la machine supporte des opérateurs de comparaison, le code est trivial : pour  $x < y$  :

$$cmp\_LT \ r_x, r_y \Rightarrow r_1$$

Les opérations de contrôle de flot peuvent alors se faire à partir d'une opération de branchement conditionnel simple :

$$cbr \ r_1 \rightarrow L_1, L_2$$

Dans le codage par position, la comparaison va mettre à jour un registre de code condition (*condition code register*), qui sera utilisé dans une instruction de branchement conditionnel (Iloc possède les deux possibilités). Dans ce cas, le code généré pour  $x < y$  (s'il est utilisé comme un booléen dans une expression booléenne) est :

$$\begin{array}{l} \text{comp} \quad r_x, r_y \Rightarrow cc_1 \\ \text{cbr\_LT} \quad cc_1 \rightarrow L_1, L_2 \\ L_1 : \text{loadI} \quad true \Rightarrow r_2 \\ \quad \text{jumpI} \quad \quad \rightarrow L_3 \\ L_2 : \text{loadI} \quad false \Rightarrow r_2 \\ \quad \text{jumpI} \quad \quad \rightarrow L_3 \\ L_3 : \text{nop} \end{array}$$

Ce type de codage permet d'optimiser les instructions du type

$$\begin{array}{l} \text{If} \quad (x < y) \\ \quad \text{then} \text{ statement}_1 \\ \quad \text{else} \text{ statement}_2 \end{array}$$

à condition que le compilateur sache les reconnaître.

L'encodage par position peut paraître moins intuitif a priori, mais en fait, il permet d'éviter l'assignation de valeur booléenne tant qu'une assignation n'est pas explicitement nécessaire. Par exemple, considérons l'expression :  $(a < b) \vee (c > d) \wedge (e < f)$ . Un générateur de code avec encodage par position produirait :

$$\begin{array}{l} \text{comp} \quad r_a, r_b \Rightarrow cc_1 \\ \text{cbr\_LT} \quad cc_1 \rightarrow L_3, L_1 \\ L_1 : \text{comp} \quad r_d, r_c \Rightarrow cc_2 \\ \quad \text{cbr\_LT} \quad cc_2 \rightarrow L_2, L_4 \\ L_2 : \text{comp} \quad r_e, r_f \Rightarrow cc_3 \\ \quad \text{cbr\_LT} \quad cc_3 \rightarrow L_3, L_4 \\ L_3 : \text{loadI} \quad true \Rightarrow r_1 \\ \quad \text{jumpI} \quad \quad \rightarrow L_5 \\ L_4 : \text{loadI} \quad false \Rightarrow r_1 \\ \quad \text{jumpI} \quad \quad \rightarrow L_5 \\ L_5 : \text{nop} \end{array}$$

L'encodage par position permet de représenter la valeur d'une expression booléenne implicitement dans le flot de contrôle du programme. La version en représentation par valeur serait :

$$\begin{array}{l} \text{cmp\_LT} \quad r_a, r_b \Rightarrow r_1 \\ \text{cmp\_GT} \quad r_c, r_d \Rightarrow r_2 \\ \text{cmp\_LT} \quad r_e, r_f \Rightarrow r_3 \\ \text{and} \quad \quad r_2, r_3 \Rightarrow r_4 \\ \text{or} \quad \quad r_1, r_4 \Rightarrow r_5 \end{array}$$

Elle est plus compacte (5 instructions au lieu de 10) mais pourra être plus longue à l'exécution. Ici elle ne l'est pas, mais on voit comment elle peut l'être : deux opérations par profondeur de l'arbre au lieu d'une opération par noeud de l'arbre.

Dans le mode de représentation par position, on peut optimiser le code produit par une technique appelée évaluation court-circuit (*short circuit evaluation*) on peut calculer la manière qui

demandera le moins d'évaluation pour décider du résultat (par exemple en C :  $(x != 0 \ \&\& \ y/x > 0.01)$  n'évaluera pas la deuxième condition si  $x$  vaut 0).

En plus de l'encodage choisi, l'implémentation des opérations de relation va dépendre fortement de l'implémentation des opérateurs de relation sur la machine cible. Voici quatre solutions envisageables avec leur expression en Iloc pour l'affectation :  $x \leftarrow (a < b) \wedge (c < d)$ . Pour faire une comparaison complète, il faudrait comparer les codes sur une opération de contrôle de flot (type `if-then-else`).

- Code condition direct. L'opérateur de condition va mettre à jour un registre de code condition (*code condition register*), la seule instruction qui peut lire ce registre est le branchement conditionnel (avec six variantes :  $<$ ,  $\leq$ ,  $=$ , etc.). Si le résultat de la comparaison est utilisé, on doit le convertir explicitement en booléen. Dans tous les cas, le code contient au moins un branchement conditionnel par opérateur de comparaison. L'avantage vient du fait que le registre de code condition peut être mis à jour par l'opération arithmétique utilisée pour faire la comparaison (différence entre les deux variables en général) ce qui permet de ne pas effectuer la comparaison dans certains cas.

```

                comp   ra, rb  ⇒  cc1
                cbr_LT cc1    →  L1, L2
L1 :  comp   rc, rd  ⇒  cc2
                cbr_LT cc2    →  L3, L2
L2  load   false ⇒  rx
                jumpI          →  L4
L3  load   true  ⇒  rx
                jumpI          →  L4
L4  nop

```

- Move conditionnel. On possède une instruction du type

```
i2i_LT cci, ra, rb ⇒ ri
```

Le move conditionnel s'exécute en un seul cycle et ne casse pas les blocs de base, il est donc plus efficace qu'un saut.

```

comp   ra, rb          ⇒  cc1
i2i_LT cc1, rtrue, rfalse ⇒  r1
comp   rc, rd          ⇒  cc2
i2i_LT cc2, rtrue, rfalse ⇒  r2
and    r1, r2          ⇒  rx

```

- Comparaison à valeurs booléennes. On supprime le registre de code condition complètement, l'opérateur de comparaison retourne un booléen dans un registre général (ou dédié aux booléens). Il y a une uniformité entre les booléens et les valeurs relationnelles.

```

cmp_LT ra, rb  ⇒  r1
cmp_LT rc, rd  ⇒  r2
and    r1, r2  ⇒  rx

```

- Exécution prédiquée. Certaines architectures autorisent le code prédiqué (ou prédicat) c'est à dire que chaque instruction du langage machine peut être précédée d'un prédicat (i.e. un booléen stocké dans un certain registre), l'instruction ne s'exécute que si le prédicat est vrai. En Iloc, cela donne :

```
(r1)? add r1, r2 ⇒ rc
```

Ici, pour l'exemple, il n'y a pas d'opération de contrôle de flot. Le code est le même que pour les comparaisons de valeurs booléennes.

```

cmp_LT ra, rb  ⇒  r1
cmp_LT rc, rd  ⇒  r2
and    r1, r2  ⇒  rx

```

### 7.3 Désucrage des opérations de contrôle de flot

**Conditionnelle** On a vu que l'on avait quelquefois le choix pour l'implémentation d'un `if-then-else` d'utiliser soit la prédication soit des branchements. La prédication permet de calculer deux instructions en parallèle et de n'utiliser qu'un résultat suivant la valeur du prédicat. Pour les petits blocs de base cela peut valoir le coup mais si les blocs de base sont grands on risque de perdre du parallélisme. Par exemple, sur une machine pouvant exécuter deux instructions en parallèle :

	<i>Unite1</i>	<i>unite2</i>		<i>Unite1</i>	<i>unite2</i>
	<i>comparaison</i> $\Rightarrow r_1$			<i>comp.andbranch.</i>	
	$(r_1) Op_1$	$\neg(r_1) Op_7$	<i>L1 :</i>	<i>Op<sub>1</sub></i>	<i>Op<sub>2</sub></i>
	$(r_1) Op_2$	$\neg(r_1) Op_8$		<i>Op<sub>3</sub></i>	<i>Op<sub>4</sub></i>
	$(r_1) Op_3$	$\neg(r_1) Op_9$		<i>Op<sub>5</sub></i>	<i>Op<sub>6</sub></i>
	$(r_1) Op_4$	$\neg(r_1) Op_{10}$	<i>L2 :</i>	<i>jumpI L<sub>out</sub></i>	
	$(r_1) Op_5$	$\neg(r_1) Op_{11}$		<i>Op<sub>7</sub></i>	<i>Op<sub>8</sub></i>
	$(r_1) Op_6$	$\neg(r_1) Op_{12}$		<i>Op<sub>9</sub></i>	<i>Op<sub>10</sub></i>
				<i>Op<sub>11</sub></i>	<i>Op<sub>12</sub></i>
				<i>jumpI L<sub>out</sub></i>	
				<i>L<sub>out</sub> : nop</i>	

Le choix de compilateur peut aussi prendre en compte le fait que les branches `then` et `else` sont déséquilibrées et qu'il peut y avoir d'autres opérations de contrôle à l'intérieur.

**Boucles** L'implémentation d'une boucle (`For`, `Do`, `While`) se fait selon une variation du schéma suivant (se reporter au document donnant la sémantique du langage) :

1. évaluer l'expression contrôlant la boucle
2. si elle est fausse, brancher après la fin de la boucle, sinon commencer le corps de boucle
3. à la fin du corps de boucle, réévaluer l'expression de contrôle
4. si elle est vraie, brancher au début de la boucle sinon continuer sur l'instruction suivante

Il y a donc un branchement par exécution du corps de boucle. En général un branchement produit une certaine latence (quelques cycles).

Exemple de génération de code pour une boucle `for` :

		<i>loadI</i>	1	$\Rightarrow$	<i>r<sub>1</sub></i>
		<i>loadI</i>	1	$\Rightarrow$	<i>r<sub>2</sub></i>
		<i>loadI</i>	100	$\Rightarrow$	<i>r<sub>3</sub></i>
		<i>cmp_LT</i>	<i>r<sub>1</sub>, r<sub>3</sub></i>	$\Rightarrow$	<i>r<sub>4</sub></i>
<code>for (i = 1; i &lt;= 100; i++)</code>	<i>L<sub>1</sub> :</i>	<i>cbr</i>	<i>r<sub>4</sub></i>	$\rightarrow$	<i>L<sub>1</sub>, L<sub>2</sub></i>
{ corps de la boucle					<i>corps de la boucle</i>
}					
		<i>add</i>	<i>r<sub>1</sub>, r<sub>2</sub></i>	$\Rightarrow$	<i>r<sub>1</sub></i>
		<i>cmp_LT</i>	<i>r<sub>1</sub>, r<sub>3</sub></i>	$\Rightarrow$	<i>r<sub>6</sub></i>
	<i>L<sub>2</sub> :</i>	<i>cbr</i>	<i>r<sub>6</sub></i>	$\rightarrow$	<i>L<sub>1</sub>, L<sub>2</sub></i>
					<i>suite</i>

On peut aussi, à la fin de la boucle, renvoyer sur le test de la condition du début mais cela crée une boucle avec plusieurs blocs de base qui sera plus difficile à optimiser. On choisira donc plutôt ce schéma là à moins que la taille du code ne soit très critique. Les boucles `while` ou `until` se compilent de la même façon (duplication du test).

Dans certains langages (comme Lisp), les itérations sont implémentées par une certaine forme de récursion : la récursion terminale (*tail recursion*). Si la dernière instruction du corps d'une fonction est un appel à cette fonction, l'appel est du type *récursion terminale*. Par exemple considérons la fonction Lisp suivante :

(define (last alon)

```

(cond
  ((empty ? alon) empty)
  ((empty ? (rest alon)) (first alon))
  (else (last (rest alon))))

```

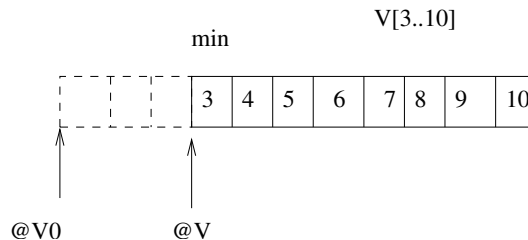
Lorsqu'on arrive à l'appel récursif, le résultat de l'appel en cours sera le résultat de l'appel récursif, tous les calculs de l'appel en cours ont été faits, on n'a donc plus besoin de l'AR de l'appel en cours. L'appel récursif peut être exécuté sans allouer un nouvel AR mais simplement en *recyclant* l'AR de l'appel en cours. On remplace le code d'un appel de procédure par un simple jump au début du code de la procédure.

**Case aka switch** La difficulté d'implémentation des instructions *case* (e.g. *switch* en C) est de sélectionner la bonne branche efficacement. Le moyen le plus simple est de faire une passe linéaire sur les conditions comme si c'était une suite imbriquée de *if-then-else*. Dans certains cas, les différents tests de branchement peuvent être ordonnés (test d'une valeur par exemple), dans ce cas on peut faire une recherche dichotomique du branchement recherché ( $O(\log(n))$ ) au lieu de  $O(n)$ .

## 7.4 Tableaux

Le stockage et l'accès aux tableaux sont extrêmement fréquents et nécessitent donc une attention particulière. Commençons par la référence à un simple vecteur (tableau à une dimension). Considérons que  $V$  a été déclaré par  $V[\text{min} \dots \text{max}]$ . Pour accéder à  $V[i]$ , le compilateur devra calculer l'*offset* de cet élément du tableau par rapport à l'adresse de base à partir de laquelle il est stocké. L'offset est  $(i - \text{min}) \times w$  ou  $w$  est la taille des éléments de  $w$ . Si  $\text{min} = 0$  et  $w$  est une puissance de deux, le code généré s'en trouvera simplifié (les multiplications par des puissance de 2 se font par décalage de bits, ce qui est en général plus rapide qu'une multiplication).

Si la valeur de  $\text{min}$  est connue à la compilation, le compilateur peut calculer l'adresse (virtuelle)  $@V_0$  qu'aurait le tableau s'il commençait à 0 (on appelle quelquefois cela le faux zéro), cela évite une soustraction à chaque accès aux tableaux :



Si la valeur de  $\text{min}$  n'est pas connue à la compilation, le faux zéro peut être calculé lors de l'initialisation du tableau. On arrive alors à un code machine pour accéder à  $V[i]$  :

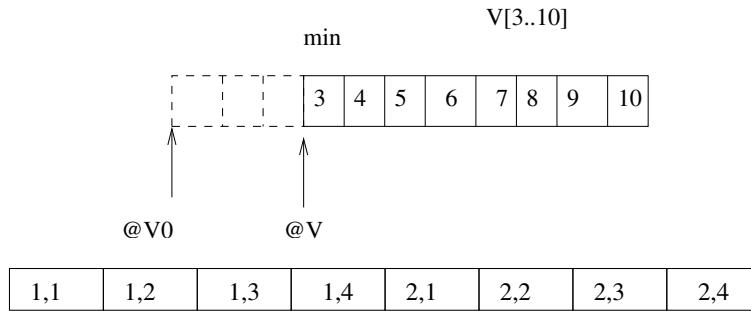
```

loadI   @V0   => r1           // faux zéro pour V
lshift  r_i,2 => r2           // i x taille des éléments
loadA0  r1,r2 => r2           // valeur de V[i]

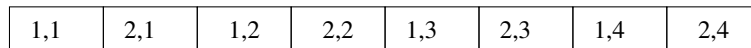
```

Pour les tableaux multidimensionnels, il faut choisir la façon d'envoyer les indices dans la mémoire. Il y a essentiellement trois méthodes, les schémas par lignes et par colonnes et les tableaux de vecteurs.

Par exemple, le tableau  $A[1..2, 1..4]$  comporte deux lignes et quatre colonnes. S'il est rangé par lignes (*row major order*), on aura la disposition suivante en mémoire :



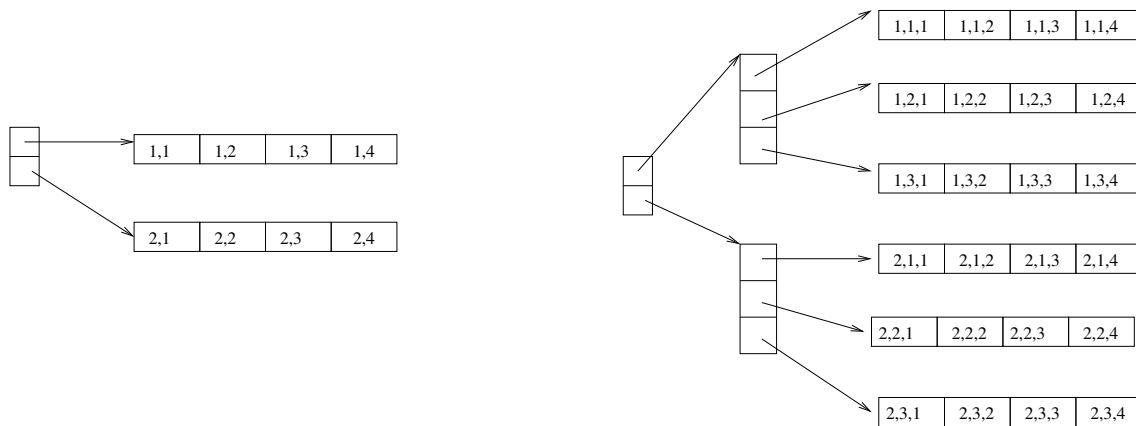
S'il est rangé par colonne on aura :



Lors d'un parcours du tableau, cet ordre aura de l'importance. En particulier si l'ensemble du tableau ne tient pas dans le cache. Par exemple avec la boucle suivante, on parcourt les cases par lignes ; si l'on intervertit les boucles  $i$  et  $j$ , on parcourera par colonnes :

```
for i ← 1 to 2
  for j ← 1 to 4
    A[i, j] ← A[i, j] + 1
```

Certains langages (java) utilisent un tableau de pointeurs sur des vecteurs. Voici le schéma pour  $A[1..2, 1..4]$  et  $B[1..2, 1..3, 1..4]$



Ce schéma est plus gourmand en place (la taille des vecteurs de pointeur grossit quadratiquement avec la dimension) et l'accès n'est pas très rapide, de plus le programmeur doit écrire une boucle pour allouer le tableau ligne par ligne. Concernant la place, ce schéma permet toutefois d'implanter efficacement des tableaux non rectangulaires (matrices triangulaires) mais c'est sans doute anecdotique.

Considérons un tableau  $A[\min_1..max_1, \min_2..max_2]$  rangé par ligne pour lequel on voudrait accéder à l'élément  $A[i, j]$ . Le calcul de l'adresse est :  $@A[i, j] = @A + (i - \min_1) \times (max_2 - \min_2 + 1) \times w + (j - \min_2) \times w$ , ou  $w$  est la taille des données du tableau. Si on nomme  $long_2 = (max_2 - \min_2 + 1)$  et que l'on développe on obtient :  $@A[i, j] = @A + i \times long_2 \times w + j \times w - (\min_1 \times long_2 \times w + \min_2 \times w)$ . On peut donc aussi précalculer un faux zéro  $@A_0 = @A - (\min_1 \times long_2 \times w + \min_2 \times w)$  et accéder l'élément  $A[i, j]$  par  $@A_0 + (i \times long_2 + j) \times w$ . Si les bornes et la taille ne sont pas connues à la compilation, on peut aussi effectuer ces calculs à l'initialisation. Les mêmes optimisations sont faites pour les tableaux rangés par colonnes.

L'accès aux tableaux utilisant des pointeurs de vecteurs nécessite simplement deux instructions par dimension (chargement de la base de la dimension  $i$ , offset jusqu'à l'élément dans la

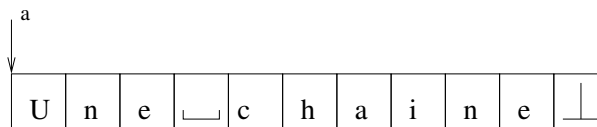
dimension  $i$ ). Pour les machines où l'accès à la mémoire était rapide comparé aux opérations arithmétiques, cela valait le coup (ordinateur avant 1985).

Lorsqu'on passe un tableau en paramètre, on le passe généralement par référence, même si dans le programme source, il est passé par valeurs. Lorsqu'un tableau est passé en paramètre à une procédure, la procédure ne connaît pas forcément ses caractéristiques (elles peuvent changer suivant les appels comme en C). Pour cela le système a besoin d'un mécanisme permettant de récupérer ces caractéristiques (dimensions, taille). Cela est fait grâce au descripteur de tableau (*dope vector*). Un descripteur de tableau contient en général un pointeur sur le début du tableau et les tailles des différentes dimensions. Le descripteur a une taille connue à la compilation (dépendant uniquement de la dimension) et peut donc être alloué dans l'AR de la procédure appelée.

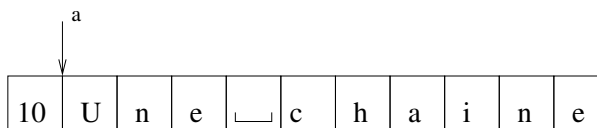
Beaucoup de compilateurs essaient de détecter les accès en dehors des bornes des tableaux. La méthode la plus simple est de faire un test sur les indices avant chaque appel à l'exécution. On peut faire un test moins précis en comparant l'offset calculé à la taille totale du tableau.

## 7.5 Chaînes de caractères

La manipulation des chaînes de caractères est assez différente suivant les langages. Elle utilise en général des instructions opérant au niveau des octets. Le choix de la représentation des chaînes de caractères a un impact important sur les performances de leur manipulation. On peut stocker une chaîne en utilisant un marqueur de fin de chaîne comme en C traditionnel :



ou en stockant la longueur de la chaîne au début (ou la taille effectivement occupée, si ce n'est pas la taille allouée) :



Les assembleurs possèdent des opérations pour manipuler les caractères (*loadAI*, *storeAI* par exemple pour lloc). Par exemple pour traduire l'instruction C :  $a[1] = b[2]$  où  $a$  et  $b$  sont des chaînes de caractères, on écrit :

```

loadI    @b    => r_b
cloadAI  r_b, 2 => r_2
loadI    @a    => r_a
cstoreAI r_2   => r_a, 1

```

Si le processeur ne possède pas d'instruction spécifique pour les caractères, la traduction est beaucoup plus délicate : en supposant que  $a$  et  $b$  commencent sur une frontière de mot (adresse en octet multiple de 4), on aura le code :

```

loadI    @b            => r_b           // adresse de b
load     r_b           => r_1           // premier mot de b
andI     r_1, 0x0000FF00 => r_2           // masque les autres caractères
lshiftI  r_2, 8        => r_3           // déplacement du caractère
loadI    @a            => r_a           // adresse de a
load     r_a           => r_4           // premier mot de a
andI     r_4, 0xFF00FFFF => r_5           // masque le deuxième caractère
or       r_3, r_5      => r_6           // on rajoute le caractère de b
storeAI  r_6, r_a, 0   => r_a, 0       // on remet le premier mot de a en mémoire

```

Pour les manipulations de chaînes (comme pour les tableaux d'ailleurs), beaucoup d'assembleurs possèdent des instructions de chargement et de stockage avec autoincrément (pré ou post), c'est à dire que l'instruction incrémente l'adresse servant à l'accès en même temps qu'elle fait l'accès, ce qui permet de faire directement l'accès suivant au mot (resp. caractère suivant) juste après. On peut aussi choisir d'utiliser des instructions sur les mots (32 bits) tant que l'on arrive pas à la fin de la chaîne.

## 7.6 Structures

Les structures sont utilisées dans de nombreux langages. Elles sont très souvent manipulées à l'aide de pointeurs et créent donc de nombreuses valeurs ambiguës. Par exemple, pour faire une liste en C, on peut déclarer les types suivants :

```

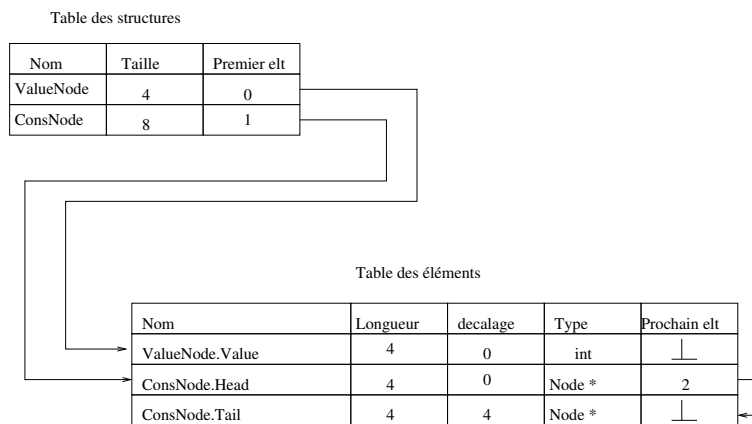
struct ValueNode {
    int Value
};

struct ConsNode {
    Node *Head;
    Node *Tail;
};

Union Node{
    struct ValueNode;
    struct ConsNode;
};

```

Afin d'émettre du code pour une référence à une structure, le compilateur doit connaître l'adresse de base de l'objet ainsi que le décalage et la longueur de chaque élément. Le compilateur construit en général une table séparée comprenant ces informations (en général une table pour les structures et une table pour les éléments).



Avec cette table, le compilateur génère du code. Par exemple pour accéder à  $p1 \rightarrow \text{Head}$  on peut générer l'instruction :

$$\text{loadA0 } r_{p1,0} \Rightarrow r_2 \quad // \text{ 0 est le décalage de 'Head'}$$

Si un programme déclare un tableau de structure dans un langage dans lequel le programmeur ne peut pas avoir accès à l'adresse à laquelle est stockée la structure, le compilateur peut choisir de stocker cela sous la forme d'un tableau dont les éléments sont des structures ou sous la forme d'une structure dont les éléments sont des tableaux. Les performances peuvent être très différentes, là encore à cause du cache.

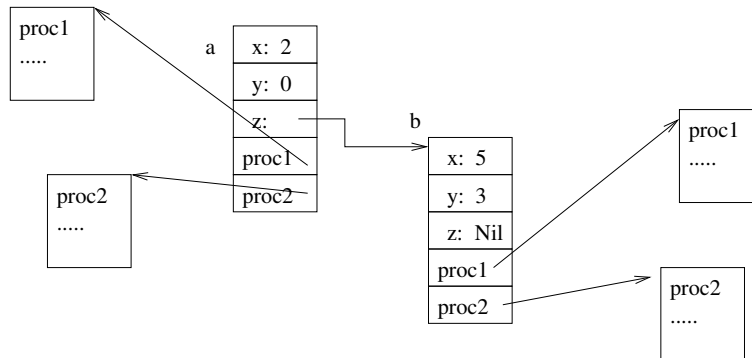
## 8 Implémentation des langages objet

Fondamentalement, l'orientation objet est une réorganisation de l'espace des noms du programme d'un schéma centré sur les procédures vers un schéma centré sur les données. En pratique, les "langages orientés objet" ne sont pas tous basés sur le même principe. Dans cette section, on présente l'implémentation d'un cas particulier de langage orienté objet.

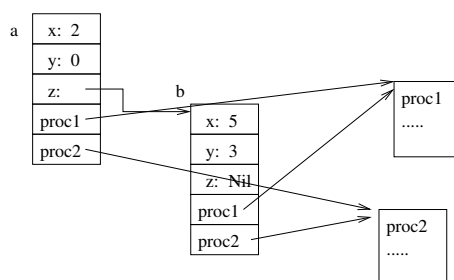
### 8.1 Espace des noms dans les langages objets

Dans les langages traditionnels, les portées sont définies à partir du code (en entrant dans une procédure ou un bloc, etc.). Dans les langages objets, les portées des noms sont construites à partir des données. Ces données qui gouvernent les règles de nommage sont appelées des objets (tout n'est pas objet, il y a aussi des variables locales aux procédures qui sont implémentées comme dans les langages classiques).

Pour le concepteur de compilateur, les objets nécessitent des mécanismes supplémentaires à la compilation et à l'exécution. Un objet est une abstraction qui possède plusieurs membres. Ces membres peuvent être des données, du code qui manipule ces données ou d'autres objets. On peut donc représenter un objet comme une structure (avec la convention que les éléments peuvent être des données, des procédures ou d'autres objets).

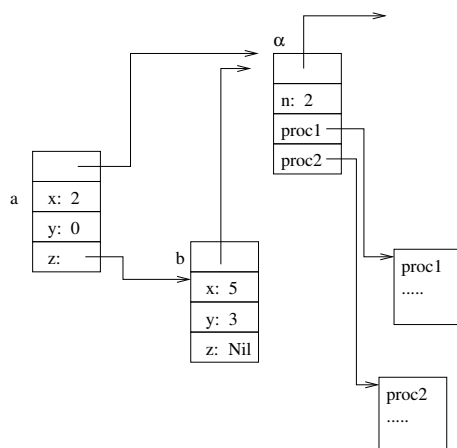


Dans le schéma ci-dessus, on voit deux objets appelés *a* et *b*. On voit que les deux variables (*x* et *y*) de *a* et *b* sont des variables différentes. En revanche, il utilisent les mêmes procédures. Comme on a représenté l'accès aux codes par des pointeurs, on pourrait imaginer qu'il partagent un même code :



Mais la manipulation de code avec des pointeurs (mélange du code et des données) pourra assez vite amener des situations inextricables, c'est pour cela que l'on introduit la notion de classe. Une classe est une abstraction qui groupe ensemble des objets similaires. Tous les objets d'une même classe ont la même structure. Tous les objets d'une même classe utilisent les mêmes portions de code. Mais chaque objet de la classe a ses propres variables et objets membres. Du point de vue de l'implémentation, on va pouvoir stocker les portions de code d'une classe dans un nouvel objet qui représente la classe :



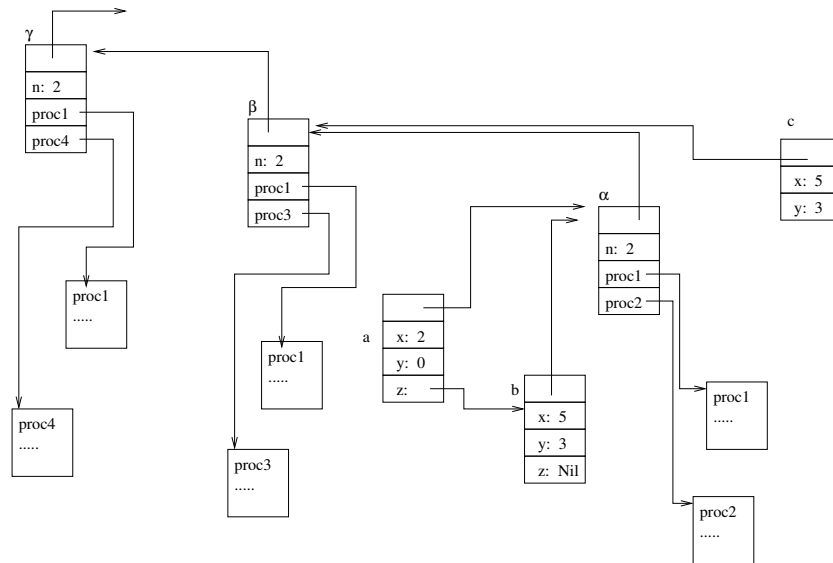


Ici  $\alpha$  est la classe à laquelle appartiennent  $a$  et  $b$ . Les objets  $a$  et  $b$  ont un nouveau champ qui représente la classe à laquelle ils appartiennent (la classe étant un objet, elle a aussi ce nouveau champ). Attention, ces diagrammes sont des représentations de l'implémentation de ces objets, pas de leur déclaration dans le langage source. Dans la déclaration de la classe  $\alpha$  on trouvera la déclaration des variables  $x$  et  $y$ . On peut maintenant introduire la terminologie utilisée pour les langages objets :

- *instance* Une instance est un objet appartenant à une certaine classe (c'est en fait la notion intuitive lié au mot objet). C'est un objet "logiciel" (tel qu'on le rencontre dans le code source).
- *enregistrement d'objet* Un enregistrement d'objet (*object record*) est la représentation concrète d'une instance (son implémentation dans la mémoire par exemple).
- *variable d'instance* Les données membres d'un objet sont des variables d'instance (exemple, la variable d'instance  $x$  de  $a$  vaut 2, en java on appelle cela les champs-*field*).
- *méthode* Une méthode est une fonction ou procédure commune à tous les objets d'une classe. C'est une procédure classique. La différence avec les langages classiques tient dans la manière de la nommer : elle ne peut être nommée qu'à partir d'un objet. En java, on ne la nomme pas  $proc_1$  mais  $x.proc_1$  où  $x$  est une instance d'une classe qui implémente  $proc_1$ .
- *récepteur* Comme les procédures sont invoquées depuis un objet, chaque procédure possède en plus un paramètre implicite qui est un pointeur sur l'enregistrement d'objet qui l'a invoqué. Dans la méthode, le récepteur est accessible par `this` ou `self`.
- *Classe* Une classe est un objet qui décrit les propriétés d'autres objets. Une classe définit les variables d'instance et les méthodes pour chaque objet de la classe. Les méthodes de la classe deviennent des variables d'instance pour les objets qui implémentent la classe.
- *variables de classe* Chaque classe est implémente par un objet. Cet objet a des variables d'instance. Ces variables sont quelquefois appelées variables de classe, elles forment un stockage persistant visible de n'importe quelle méthode de la classe et indépendant du récepteur.

La plupart des langages orientés objet implémentent la notion d'héritage. L'héritage permet de définir une classe de classes. Une classe  $\alpha$  peut avoir une classe parent  $\beta$  (ou super-classe). Toutes les méthodes de la classe  $\beta$  sont alors accessibles par un objet de la classe  $\alpha$ . Toutes les variables d'instance de  $\beta$  sont variables d'instance de  $\alpha$  et la résolution de noms des méthodes se fait récursivement (comme pour les portées des variables dans les langages classiques).

Voici un exemple d'héritage. Les objets  $a$  et  $b$  sont membres de la classe  $\alpha$  dont  $\beta$  est une super-classe. Chacune des trois classes  $\alpha$ ,  $\beta$  et  $\gamma$  implémente une méthode nommée  $proc_1$ . On voit un objet  $c$  qui est une instance de la classe  $\beta$ , les objets de cette classe ont les variables d'instance  $x$  et  $y$  mais pas  $z$ .



Si un programme invoque  $a.proc_1$ , il exécutera la procédure définie dans la classe  $\alpha$ , s'il invoque  $b.proc_4$  on exécutera la procédure définie dans la classe  $\gamma$ . Le mécanisme pour retrouver le code de la procédure appelée à l'exécution est appelé *dispatch*. Le compilateur met en place une structure équivalente à la coordonnée de distance statique : une paire  $\langle distance, decalage \rangle$ . Si la structure de la classe est statique tout cela peut être déterminé à la compilation. Si l'exécution peut changer la hiérarchie des classes et l'héritage, générer du code efficace devient beaucoup plus difficile.

Quelles sont les règles de visibilité pour une procédure appelée (par exemple  $a.proc_1$ )? D'abord,  $proc_1$  a accès à toutes les variables déclarées dans la procédure. Elle peut aussi accéder les paramètres. Cela implique qu'une méthode active possède un enregistrement d'activation (AR) comme dans les langages classiques. Ensuite,  $proc_1$  peut accéder aux variables d'instance de  $a$  qui sont stockées dans l'enregistrement d'objet de  $a$ . Ensuite, elle a accès aux variables de classe de la classe qui a défini  $proc_1$  (en l'occurrence  $\alpha$ ) ainsi qu'aux variables d'instance de ses super-classes. Les règles sont un peu différentes de celles des langages classiques, mais on voit bien que l'on peut mettre en place un mécanisme similaire à celui utilisant les coordonnées de distance statique.

Du fait du rôle central joué par les données dans les langages objets, le passage de paramètres joue un rôle plus important. Si une méthode appelle une autre méthode, elle ne peut le faire que si elle possède des objets récepteurs pour cette méthode. Typiquement, ces objets sont soit globaux soit des paramètres.

## 8.2 Génération de code

Considérons le problème de génération de code pour une méthode individuelle. Comme la méthode peut accéder tous les membres de tous les objets qui peuvent être récepteurs, le compilateur doit établir des décalages pour chacun de ces objets qui s'expriment uniformément depuis la méthode. Le compilateur construit ces décalages lorsqu'il traite les déclarations de classe (les objets eux-mêmes n'ont pas de code).

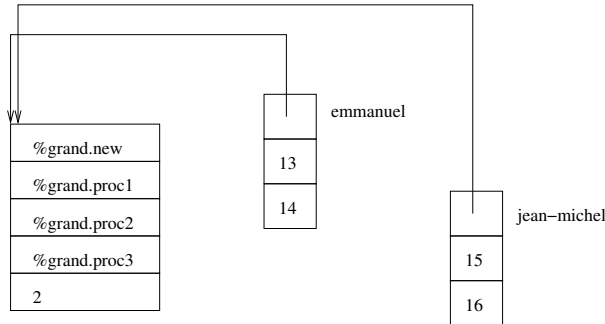
Le cas le plus simple est le cas où la structure de la classe est connue à la compilation et qu'il n'y a pas d'héritage. Considérons une classe *grand* qui possède les méthodes  $proc_1$ ,  $proc_2$ ,  $proc_3$  et les membres  $x$  et  $y$  qui sont des nombres. La classe *grand* possède une variable de classe,  $n$ , qui enregistre le nombre d'enregistrements d'objets de cette classe qui ont été créés. Pour autoriser la création d'objet, toutes les classes implémentent la méthode *new* (au décalage 0 dans la table des méthodes).

Dans ce cas, l'enregistrement d'objet pour une instance de la classe *grand* est juste un vecteur de longueur 3. La première case est un pointeur sur sa classe (*grand*), qui contient l'adresse de la représentation concrète de la classe *grand*. Les deux autres cases contiennent les données  $x$  et  $y$ . Chaque objet de la classe *grand* a le même type d'enregistrement d'objet, créé par la méthode *new*.

L'enregistrement de classe possède un espace pour ses variables de classe ( $n$  ici) et pour toutes ses méthodes. Le compilateur mettra donc en place une table de ce type pour la classe de *grand* :

	<i>new</i>	<i>proc<sub>1</sub></i>	<i>proc<sub>2</sub></i>	<i>proc<sub>3</sub></i>	<i>n</i>
<i>decalage</i>	0	4	8	12	16

Après que le code ait créé deux instances de *grand* (ici *emmanuel* et *jean-michel*) les structures à l'exécution devraient ressembler à cela :

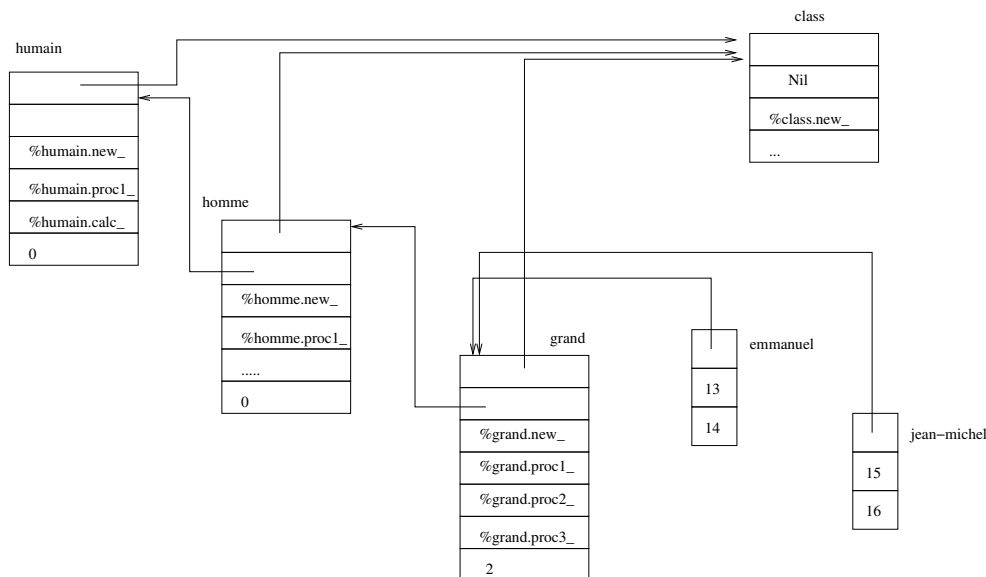


La table des méthodes de la classe *grand* contient des labels (symbole créé par le compilateur). Ces labels sont associés aux adresses auxquelles sont stockées les procédures.

Considérons maintenant le code que le compilateur devra générer lors de l'exécution de la méthode *new* d'un objet de classe *grand*. Il doit allouer la place pour le nouvel enregistrement d'objet, initialiser les données membres et retourner un pointeur sur cette nouvelle structure. La procédure doit aussi respecter les règles de l'édition de lien (sauvegarder les registres, créer un AR pour les variables locales et restaurer l'environnement avant de finir). En plus des paramètres formels, le compilateur ajoute le paramètre implicite *this*.

Lorsqu'on rencontre l'appel *jean-michel.proc<sub>1</sub>*, le code généré produit les actions suivantes : accès au décalage 0 à partir de l'adresse de *jean-michel* (adresse de la classe *grand*), chargement de l'adresse de la méthode *proc<sub>1</sub>*, et génération d'un appel de procédure à cette adresse, en mettant comme paramètre *this* l'adresse de *jean-michel*.

Le schéma est plus complexe lorsque l'héritage est autorisé. Du fait de l'héritage, chaque enregistrement de classe possède un pointeur de classe (qui pointe sur l'objet *classe*, qui implémente la méthode *new* par défaut), et un pointeur de super-classe dont elle hérite. Tant que la structure des classes est fixée, le compilateur peut résoudre les invocations de méthodes par une structure du type coordonnées de distance statique. Considérons le schéma suivant :



Les coordonnées des méthodes sont :

	<i>new</i>	<i>proc<sub>1</sub></i>	<i>proc<sub>2</sub></i>	<i>proc<sub>3</sub></i>	<i>calc</i>
<i>grand</i>	$\langle 0, 8 \rangle$	$\langle 0, 12 \rangle$	$\langle 0, 16 \rangle$	$\langle 0, 20 \rangle$	$\langle 2, 16 \rangle$
<i>homme</i>	$\langle 0, 8 \rangle$	$\langle 0, 12 \rangle$	—	—	$\langle 1, 16 \rangle$
<i>humain</i>	$\langle 0, 8 \rangle$	$\langle 0, 12 \rangle$	—	—	$\langle 0, 16 \rangle$
<i>class</i>	$\langle 0, 8 \rangle$	—	—	—	—

Lorsque *jean-michel.calc* est invoqué, le compilateur récupère le nom et l'adresse de la classe *geant* à laquelle appartient *jean-michel*. Les coordonnées de la méthode *calc* pour *jean-michel* sont  $\langle 2, 16 \rangle$ . Le compilateur génère alors le code pour remonter le long des pointeurs de super-classe jusqu'à la classe *humain*, récupère l'adresse de la procédure *calc* et appelle la procédure.

Ce schéma est simple, mais implique une condition que doit respecter le rangement des données dans les enregistrements d'objets : lorsque *jean-michel* est récepteur de la méthode *calc* de la classe *humain*, la méthode ne connaît que les variables d'instance que *jean-michel* hérite de *humain* et de ses ancêtres (pas celles héritées de la classe *homme*). Pour que les méthodes héritées fonctionnent, ces variables doivent avoir le même décalage dans un objet de classe *humain* que dans un objet de classe *grand*. Autrement la méthode désignerait, suivant le récepteur, des variables différentes avec un même décalage.

Cela suggère que les membres des méthodes soient rangés en mémoire dans l'ordre d'héritage. Les variables d'instance d'un enregistrement de la classe *grand* seront rangées de la manière suivante :

<i>pointeurs de classes</i>	<i>data humain</i>	<i>data homme</i>	<i>data grand</i>
-----------------------------	--------------------	-------------------	-------------------

La même contrainte s'applique aux enregistrements de classes. Dans la cas de l'héritage multiple, la situation est un peu plus compliquée.

## 9 Génération de code : sélection d'instructions

On a vu comment, à partir d'un arbre syntaxique, générer du code pour des structures particulières (structures de contrôle, appels de procédure, etc.). En général pour générer complètement le code, il faut :

1. Sélectionner les instructions de la machine cible à utiliser à partir de la représentation intermédiaire (sélection d'instructions, *instruction selection*).
2. Choisir un ordre d'exécution de ces instructions (ordonnancement d'instructions, *instruction scheduling*).
3. Décider quelles valeurs résideront dans les registres (allocation de registre, *register allocation*)

On reviendra sur les deux derniers traitements plus loin. Ici, le traitement qui nous intéresse est la sélection d'instructions.

### 9.1 Génération de code optimisée et recible

Nous avons déjà vu un algorithme simple et sûr qui parcourt un AST et génère le code correspondant, mais le code obtenu était très sous-optimal.

La difficulté principale du processus de sélection d'instructions est bien une question d'*optimisation* : il y a en général plusieurs manières d'implémenter un calcul donné avec le jeu d'instructions de la machine cible (*instruction set architecture, ISA*), et ces variantes auront des coûts différents. Par exemple un accès à une case d'un tableau  $A[i]$ , lorsque  $i$  est une variable en mémoire, peut être implémenté par l'assembleur Iloc suivant :

$$\begin{array}{lll} \text{loadAI} & r_{arp}, @i & \Rightarrow r_i \\ \text{mulI} & r_i, 2 & \Rightarrow r_i / * \text{ éléments du tableau de taille } 2 * / \\ \text{loadAI} & r_{arp}, @A & \Rightarrow r_A \\ \text{loadAO} & r_A, r_i & \Rightarrow r_2 \end{array}$$

Mais la multiplication par 2 peut être implémentée comme un shift, ou bien comme :

$$\text{add } r_i, r_i \Rightarrow r_i$$

Sur d'autres architectures, on aura une instruction de multiplication-accumulation qui pourra réaliser  $r_A + 2 * r_i$ .

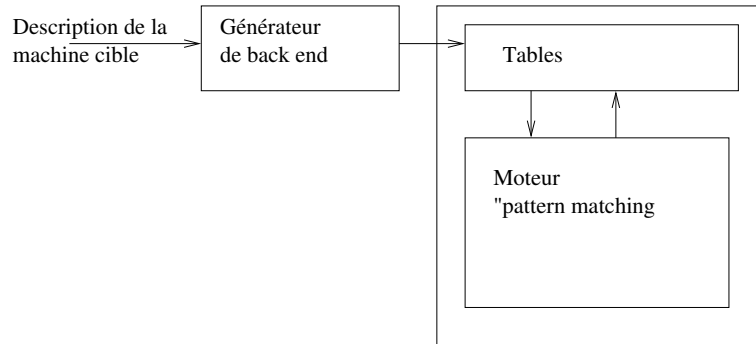
Les coûts de chaque instruction sont en général constants (entre 1 et quelques cycles), avec deux exceptions :

- Les accès mémoire dont le temps d'exécution est difficilement prédictible (à cause du cache).
- Certaines opérations arithmétiques peuvent avoir un coût qui dépend des opérandes. Par exemple, sur certains ARM, le nombre de cycles pour une multiplication dépend du multiplicateur.

En général la fonction à optimiser est la vitesse d'exécution, mais on peut imaginer d'autres fonctions (consommation électrique, taille de code, etc.). De plus, un certain nombre de contraintes spécifiques à l'architecture cible sont rajoutées : certains registres sont dédiés aux opérations sur les entiers, ou les flottants. Certaines opérations ne peuvent s'exécuter que sur certaines unités fonctionnelles.

Tout ceci reste assez facile à gérer par bricolage éclairé, jusqu'à ce qu'on veuille réaliser un compilateur *recible*, c'est-à-dire qui puisse générer du code pour différentes machines cibles (ou suivre les évolutions d'une famille de processeurs). Dans ce cas, on va rechercher une approche systématique du problème de la sélection d'instructions qui permet de construire des compilateurs facilement recibles sans avoir à réécrire la majeure partie du compilateur. Sélection et ordonnancement d'instruction et allocation de registres concernent le code assembleur de la machine cible, elles utilisent donc des informations spécifiques à la cible : par exemple, latence des opérations, nombre de registres, capacités des unités fonctionnelles, restriction d'alignement, etc. En réalité, beaucoup de ces caractéristiques sont nécessaires pour des étapes précédant la sélection d'instructions (par exemple : les restrictions d'alignement, les conventions pour les AR).

Donc on veut construire un *générateur automatique de sélecteur d'instructions* selon l'architecture suivante :



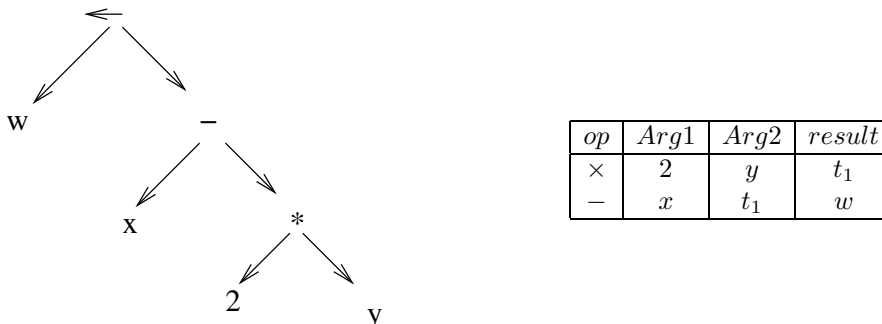
Nous allons voir deux techniques utilisées pour automatiser la construction d'un sélecteur d'instructions : la première utilise la théorie des réécritures d'arbres pour créer un système de réécriture ascendant (*bottom up rewriting system*, BURS), il fonctionne sur les IR à base d'arbre. Le second utilise une optimisation classique : l'optimisation par fenêtrage (*peephole optimization*) qui essaie de trouver, parmi une librairie de morceaux de code prédéfinis (*pattern*), la meilleure manière d'implémenter un morceau de l'IR limité par une fenêtre (fonctionne sur les IR linéaires).

## 9.2 Sélection d'instruction par parcours d'arbre (BURS)

Considérons l'instruction suivante :

$$w \leftarrow x - 2 \times y$$

On peut choisir de la représenter en représentation intermédiaire par un arbre ou une IR linéaire.



Le schéma le plus simple pour générer du code est le parcours d'arbre. Par exemple, si on ne s'intéresse qu'à la partie arithmétique  $x - 2 \times y$ , on peut utiliser la fonction *expr* suivante :

```

expr(nœud){
  int result,t1,t2;
  switch(type(nœud))
  {
    case ×, -, +, ÷ :
      t1 ← expr(fils_gauche(nœud));
      t2 ← expr(fils_droit(nœud));
      result ← NextRegister();
      emit(op(nœud), t1, t2, result);
      break;
    case Identificateur :
      t1 ← base(nœud);
      t2 ← decalage(nœud);
      result ← NextRegister();
      emit(loadAI, t1, t2, result);
      break;
  }
}

```

```

    case Number :
        result ← NextRegister();
        emit(loadI, val(nœud), result);
        break;
    }
return result
}

```

Le code généré lors du parcours de l'arbre correspondant à  $x - 2 \times y$  est :

```

loadAI  rarp, @x ⇒ rx
loadI   2      ⇒ r2
loadAI  rarp, @x ⇒ ry
mult    r2, ry ⇒ r3
sub     rx, r3 ⇒ r4

```

Mais pour avoir un code efficace il faudrait des traitements plus complexes sur les nœuds. Par exemple, voici un code qui ne serait pas traduit optimalement. Pour traduire  $a \times 2$  (en supposant que les coordonnées de distance statique de  $a$  sont  $\langle arp, 4 \rangle$ ), on aura :

```

loadAI  rarp, 4 ⇒ r1
loadI   2      ⇒ r2
mult    r1, r2 ⇒ r3

```

au lieu de

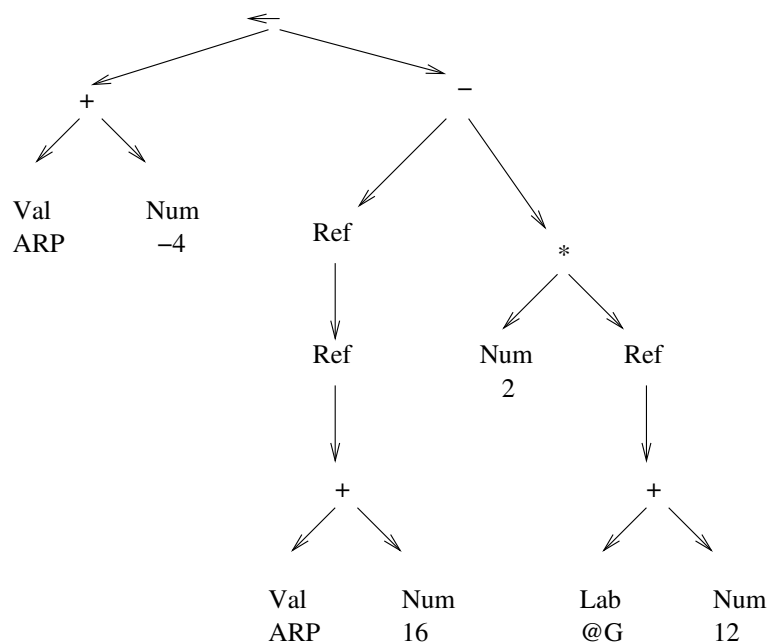
```

loadAI  rarp, 4 ⇒ r1
multI   r1, 2   ⇒ r2

```

Ce deuxième cas expose un problème intrinsèque à l'algorithme de parcours d'arbre simple. Il y a utilisation d'une information non locale au nœud (le fait que l'un des arguments de la multiplication est une constante). On pourrait aussi réutiliser localement des valeurs comme les décalages en mémoire.

**AST recyclable** On commence par raffiner l'AST en descendant à une description la plus explicite possible, n'utilisant que des opérations simples, jamais d'opérations composées. Remarquez que ce sont en gros les opérations du code trois adresse, donc il sera facile de reconstruire un tel arbre à partir d'une représentation sous forme de blocs de base, surtout en SSA. Pour notre exemple, cela pourrait donner :



Dans cet arbre, *Val* indique une valeur dont on sait qu'elle réside dans un registre (ici l'ARP). Un nœud *Lab* (pour *Label*) représente un symbole relogeable (*relocatable symbol*) c'est à dire un label de l'assembleur utilisé pour designer l'emplacement d'une donnée ou d'un symbole (par exemple, une variable statique) qui sera résolu à l'édition de lien. Un *Ref* signifie un niveau d'indirection. On voit apparaître des informations de bas niveau comme par exemple le fait que *x* est un paramètre passé par référence.

L'algorithme de réécriture d'arbre va cibler une représentation en arbre de l'assembleur cible (ici Iloc). On utilisera aussi une notation pour décrire les arbres en utilisant une notation préfixe. Par exemple l'arbre ci-dessus se décrira :

$$Gets(+ (Val_1, Num_1), - (Ref(Ref(+ (Val_2, Num_2))), \times (Num_3, Ref(+ (Lab_1, Num_4))))))$$

**Arbres d'opération** Les instructions de l'assembleur cible seront également décrites par des *arbres d'opérations* (expression des instructions Iloc sous forme d'arbre). Ainsi une instruction composée sera décrite par un arbre complexe. Par exemple le LoadAI sera décrit par l'arbre  $Ref(+ (Reg_2, Num_1))$ .

**Pavage d'arbre** On va essayer de recouvrir l'AST par des arbres d'opérations, on appelle cette opération un pavage (*tiling*). Un pavage de l'AST est un ensemble de couples du type (*nœud de l'AST, arbre d'opération*) qui indique que *arbre d'opération* implémente *nœud de l'AST*. Une fois que l'on a un pavage qui couvre tous les nœuds de l'arbre, on peut générer le code par un simple parcours en profondeur de l'arbre (comme on l'a fait pour l'expression arithmétique  $x - 2 \times y$ ). La difficulté est de générer rapidement un bon pavage sachant qu'une opération peut implémenter plusieurs nœuds de l'AST. En général, on associe un coût à chaque arbre d'opération et on essaie de minimiser le coût total.

**Règles de réécriture et grammaire d'arbre** On va décrire l'assembleur cible de manière formelle par un ensemble de règles de réécriture. Une règle de réécriture comporte une production exprimée sur la grammaire d'arbre, un coût, et un code produit. Par exemple, une règle qui fait la correspondance entre un sous-arbre d'addition  $+(r_1, r_2)$  et l'instruction Iloc *add* sera  $(Reg \rightarrow + (Reg_1, Reg_2), 1, \text{addr}_1, r_2 \Rightarrow r_{new})$ .

Le symbole en partie gauche correspond à un symbole non-terminal représentant un ensemble de sous arbres que la grammaire d'arbre peut générer. Les symboles  $Reg_1$  et  $Reg_2$  représentent aussi le non-terminal *Reg*, ils sont numérotés pour pouvoir les identifier (comme les \$1 et \$2 de yacc). Le nœud  $+$  ainsi décrit prend obligatoirement deux registres en argument et range son résultat dans un registre. Ces grammaires sont en général ambiguës à cause du fait qu'il y a plusieurs manières d'exprimer un même code.

Le coût associé à chaque production doit indiquer au générateur de code une estimation réaliste du coût de l'exécution du code. La plupart des techniques utilisent des coûts fixes, certaines utilisent des coûts variables (i.e. reflétant les choix faits préalablement, au cours du pavage).

**Exemple** Considérons l'ensemble possible de règles suivantes pour réécrire l'AST de bas niveau mentionné précédemment.



	règle	coût	code
1	$Goal \rightarrow Assign$	0	
2	$Assign \rightarrow Gets(Reg_1, Reg_2)$	1	$store \quad r_2 \Rightarrow r_1$
3	$Assign \rightarrow Gets(+ (Reg_1, Reg_2), Reg_3)$	1	$storeAO \quad r_3 \Rightarrow r_1, r_2$
4	$Assign \rightarrow Gets(+ (Reg_1, Num_2), Reg_3)$	1	$storeAI \quad r_3 \Rightarrow r_1, n_2$
5	$Assign \rightarrow Gets(+ (Num_1, Reg_2), Reg_3)$	1	$storeAI \quad r_3 \Rightarrow r_2, n_1$
6	$Reg \rightarrow Lab_1$	1	$loadI \quad l_1 \Rightarrow r_{new}$
7	$Reg \rightarrow Val$	0	
8	$Reg \rightarrow Num_1$	1	$loadI \quad n_1 \Rightarrow r_{new}$
9	$Reg \rightarrow Ref(Reg_1)$	1	$load \quad r_1 \Rightarrow r_{new}$
10	$Reg \rightarrow Ref(+ (Reg_1, Reg_2))$	1	$loadAO \quad r_1, r_2 \Rightarrow r_{new}$
11	$Reg \rightarrow Ref(+ (Reg_1, Num_2))$	1	$loadAI \quad r_1, n_2 \Rightarrow r_{new}$
12	$Reg \rightarrow Ref(+ (Num_1, Reg_2))$	1	$loadAI \quad r_2, n_1 \Rightarrow r_{new}$
13	$Reg \rightarrow + (Reg_1, Reg_2)$	1	$add \quad r_1, r_2 \Rightarrow r_{new}$
14	$Reg \rightarrow + (Reg_1, Num_2)$	1	$addI \quad r_1, n_2 \Rightarrow r_{new}$
15	$Reg \rightarrow + (Num_1, Reg_2)$	1	$addI \quad r_2, n_1 \Rightarrow r_{new}$
16	$Reg \rightarrow - (Reg_1, Reg_2)$	1	$sub \quad r_1, r_2 \Rightarrow r_{new}$
17	$Reg \rightarrow - (Reg_1, Num_2)$	1	$subI \quad r_1, n_2 \Rightarrow r_{new}$
18	$Reg \rightarrow - (Num_1, Reg_2)$	1	$subI \quad r_2, n_1 \Rightarrow r_{new}$
19	$Reg \rightarrow \times (Reg_1, Reg_2)$	1	$mult \quad r_1, r_2 \Rightarrow r_{new}$
20	$Reg \rightarrow \times (Reg_1, Num_2)$	1	$multI \quad r_1, n_2 \Rightarrow r_{new}$
21	$Reg \rightarrow \times (Num_1, Reg_2)$	1	$multI \quad r_2, n_1 \Rightarrow r_{new}$
22	$Num \rightarrow Lab_1$	0	

Notons que la grammaire proposée est ambiguë (par exemple, la règle 11 peut être générée en combinant les règles 14 et 9).

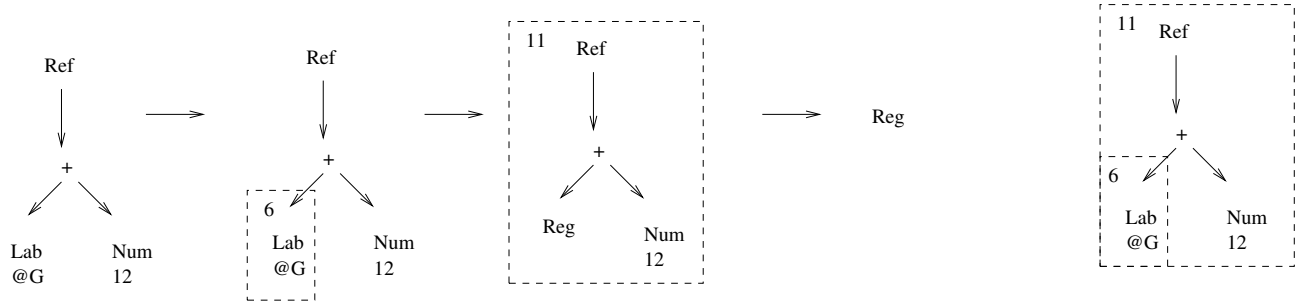
Il faut remarquer que certains symboles (par exemple  $Reg$ ) sont à la fois terminaux et non-terminaux.

**Type de stockage** Pour comprendre comment utiliser ces règles, une notion importante est le *type de stockage* de chaque valeur. Ici, on a deux types de stockage :  $Reg$  pour des registres et  $Num$  pour des valeurs ( $Lab$  est considéré comme une valeur, pour éviter ce typage supplémentaire, on pourrait rajouter une règle gratuite  $Num \rightarrow Lab$ . Un ensemble de règle plus complet comprendra la distinction entre registre et case mémoire).

À chaque règle  $R_1$ , on peut associer un type de stockage pour la partie gauche et un type de stockage pour chacun des arguments de la partie droite. Pour l'algorithme mentionné plus loin, on appellera respectivement  $left(R_1)$ ,  $Arg1(right(R_1))$ ,  $Arg2(right(R_1))$  ces types de stockage. Par exemple, pour  $R_1$  la règle  $Reg \rightarrow + (Num, Reg)$ , on a  $Arg1(right(R_1))$  qui vaut  $Num$  et  $Arg2(right(R_1))$  qui vaut  $Reg$ .

La règle que nous avons utilisée pour l'addition dans la traduction simple était la règle 13. On voit que l'on a des règles plus complexes qui décrivent des schémas de plusieurs niveaux de nœuds. Par exemple :  $Ref(+ (Reg, Num))$ . Découvrir un tel schéma dans l'arbre ne peut pas se faire avec un simple parcours en profondeur. Pour appliquer ces règles sur un arbre, on va chercher une séquence de réécriture qui réduira l'arbre à un simple symbole.

Par exemple, pour le sous-arbre  $Ref(+ (@G, 12))$ , on peut trouver la séquence  $\langle 6, 11 \rangle$  qui effectuera la réduction suivante (que l'on peut résumer en un simple schéma, ici à droite) :



Pour cet exemple particulier, nous avons d'autres possibilités : les séquences  $\langle 22, 8, 12 \rangle$  (car *Lab* à un stockage de type *Num*) qui a un coût 2 également, les séquences  $\langle 6, 8, 10 \rangle$ ,  $\langle 22, 8, 6, 10 \rangle$ ,  $\langle 6, 14, 9 \rangle$  et  $\langle 8, 15, 9 \rangle$  ont un coût 3 et les séquences  $\langle 6, 8, 13, 9 \rangle$  et  $\langle 22, 8, 6, 13, 9 \rangle$  ont un coût 4 :

$loadI$	$@G$	$\Rightarrow r_i$	$loadI$	$12$	$\Rightarrow r_i$	$loadI$	$@G$	$\Rightarrow r_i$
$loadAI$	$r_i, 12$	$\Rightarrow r_j$	$loadAI$	$r_i, @G$	$\Rightarrow r_j$	$loadI$	$12$	$\Rightarrow r_j$
$\langle 6, 11 \rangle$			$\langle 22, 8, 12 \rangle$			$loadAO$	$r_i, r_j$	$\Rightarrow r_k$
						$\langle 6, 8, 10 \rangle$		
$loadI$	$12$	$\Rightarrow r_i$	$loadI$	$@G$	$\Rightarrow r_i$	$loadI$	$12$	$\Rightarrow r_i$
$loadI$	$@G$	$\Rightarrow r_j$	$addI$	$r_i, 12$	$\Rightarrow r_j$	$addI$	$r_i, @G$	$\Rightarrow r_j$
$loadAO$	$r_i, r_j$	$\Rightarrow r_k$	$load$	$r_j$	$\Rightarrow r_j$	$load$	$r_j$	$\Rightarrow r_k$
$\langle 8, 6, 10 \rangle$			$\langle 6, 14, 9 \rangle$			$\langle 22, 8, 15, 9 \rangle$		
$loadI$	$@G$	$\Rightarrow r_i$	$loadI$	$12$	$\Rightarrow r_i$			
$loadI$	$12$	$\Rightarrow r_j$	$loadI$	$@G$	$\Rightarrow r_j$			
$add$	$r_i, r_j$	$\Rightarrow r_k$	$add$	$r_i, r_j$	$\Rightarrow r_k$			
$load$	$r_k$	$\Rightarrow r_l$	$load$	$r_k$	$\Rightarrow r_l$			
$\langle 6, 8, 13, 9 \rangle$			$\langle 8, 6, 13, 9 \rangle$					

**Comment trouver un pavage (approche naïve)** L'idée derrière les BURS (*bottom up rewriting system*) est la suivante : par un parcours en profondeur remontant (*post-order*, ou *bottom up*), on va associer à chaque nœud de l'arbre un ensemble de règles qui peuvent être utilisées pour réécrire le nœud. On pourra alors par parcours en profondeur descendant (*pre-order*, ou *top down*) générer la meilleure instruction à chaque nœud.

Définissons l'*étiquette* d'un nœud de l'AST comme un ensemble de règles de réécriture. Par exemple on a vu que le nœud  $+$  dans  $+_1(Lab_1, Num_1)$  peut utiliser les règles 13, 14 ou 15 suivant les règles que l'on avait utilisées pour ses fils (on a mis des indices pour bien signifier qu'il s'agit d'une instance particulière, un morceau de l'AST). Donc l'étiquette du nœud  $+$  peut être composée de ces trois règles :  $\{Reg \rightarrow +(Reg, Reg), Reg \rightarrow +(Reg, Num), Reg \rightarrow +(Num, Reg)\}$ . On peut restreindre le nombre de règles applicables si l'on connaît les fils du nœud  $+$  (et les règles applicables sur ses fils). Par exemple si l'on a  $+_2(Reg_1, Reg_2)$ , comme il n'existe pas de règle transformant un résultat de type registre en un résultat de type valeur, on sait que la seule règle qui peut s'appliquer est  $Reg \rightarrow +(Reg, Reg)$ .

On note  $Rules(n)$  l'étiquette d'un nœud. Par extension on note  $left(Rules(n))$  l'ensemble des type de stockage des parties gauches des règles contenues dans l'étiquette (ici  $left(Rules(+_1)) = \{Reg\}$ ). Pour simplifier l'algorithme, on va supposer que toutes les opérations ont au plus deux arguments et que les parties droite des règles ne possèdent qu'une opération. Ce n'était pas le cas de notre grammaire d'arbre, mais on peut toujours la transformer en une grammaire équivalente en rajoutant des non-terminaux intermédiaires, par exemple :

10 :	$Reg \rightarrow Ref(+ (Reg, Reg))$	devient	$Reg \rightarrow Ref(R10P2)$	$R10P2 \rightarrow +(Reg, Reg)$
11 :	$Reg \rightarrow Ref(+ (Reg, Num))$	devient	$Reg \rightarrow Ref(R11P2)$	$R11P2 \rightarrow +(Reg, Num)$
12 :	$Reg \rightarrow Ref(+ (Num, Reg))$	devient	$Reg \rightarrow Ref(R12P2)$	$R12P2 \rightarrow +(Num, Reg)$

Dans ce cas on choisit de laisser le coût total de la règle initiale sur la première des règles réécrites.

Voici l'algorithme qui permet de calculer les étiquettes d'un AST.

```

Tile(n)
  Rules(n) ← ∅
  si n est un nœud binaire
  alors
    Tile(Arg2(n))
    Tile(Arg1(n))
    Pour chaque règle r qui implémente n // i.e. si oper(r) = oper(n)
      si Arg2(right(r)) ∈ left(Rules(Arg2(n)))
        et Arg1(right(r)) ∈ left(Rules(Arg1(n))) // i.e. si les types des fils sont compatibles
        alors
          Rules(n) ← Rules(n) ∪ {r}
  sinon // (nœud unaire)
  si n est un nœud unaire
  alors
    Tile(fils(n))
    Pour chaque règle r qui implémente n // i.e. si oper(r) = oper(n)
      si fils(right(r)) ∈ left(Rules(fils(n)))
      alors
        Rules(n) ← Rules(n) ∪ {r}
  sinon // (n est une feuille)
    Rules(n) ← { toutes les règles qui implémentent n }

```

Voyons ce que donne cet algorithme sur le sous-AST :  $Ref_1(+_1(Lab_1, Num_1))$ . En arrivant sur le nœud  $Lab_1$  les seules règles qui implémentent ce nœud sont les règles 6 et 22 :  $Reg \rightarrow Lab$  et  $Num \rightarrow Lab$ . On construit donc l'étiquette de ce nœud par :

$$Rules(Lab_1) = \{Lab, Reg \rightarrow Lab, Num \rightarrow Lab\}$$

De même, la seule règle implémentant le nœud  $Num$  est la règle 8 et 6 :

$$Rules(Num_1) = \{Num, Reg \rightarrow Num\}$$

En revanche, lorsqu'on remonte d'un cran (nœud  $+_1(Lab_1, Num_1)$ ), on a beaucoup plus de possibilités. On arrive à l'étiquette suivante :

$$Rules(+_1) = \left\{ \begin{array}{l} Reg \rightarrow +(Reg_1, Reg_2), Reg \rightarrow +(Reg_1, Num_2), Reg \rightarrow +(Num_1, Reg_2), \\ R10P2 \rightarrow +(Reg, Reg), R11P2 \rightarrow +(Reg, Num), R12P2 \rightarrow +(Num, Reg), \end{array} \right\}$$

et enfin pour le nœud  $Ref_1$  on a les règles 9, 10, 11, 12.

$$Rules(Ref_1) = \left\{ \begin{array}{l} Reg \rightarrow Ref(Reg_1), Reg \rightarrow Ref(R10P2), \\ Reg \rightarrow Ref(R11P2), Reg \rightarrow Ref(R12P2) \end{array} \right\}$$

**BURS, approche futée** On imagine bien que le travail que fait l'algorithme précédent est souvent répétitif. En fait, comme l'ensemble des règles est fini (soit  $R$  son cardinal), il y a également un ensemble fini d'étiquettes possibles (de cardinal au pire  $2^R$ , mais beaucoup moins en pratique car il y a une majorité d'étiquettes qui ne peuvent apparaître par l'algo précédent).

L'idée futée est de précalculer cet ensemble des étiquettes possibles, et au passage de précalculer une fonction de transition qui, en fonction de l'opération du nœud et des étiquettes de ses fils, retourne l'étiquette de ce nœud. Cela ne vous rappelle rien ? Mais si, c'est la construction d'un automate reconnaissant une grammaire à partir de la grammaire. Le rôle du *token* d'entrée est joué par l'opération du nœud.

On peut précalculer les transitions d'une étiquette à une autre dans une table de taille :

$$|\text{Operations}| \times |\text{Etiquettes}| \times |\text{Etiquettes}|$$

Comme pour les grammaires classiques, cette table est en général creuse et il existe des techniques efficaces pour la comprimer. Une fois qu'on a construit cette table, l'algorithme qui réalise le pavage de l'arbre en remontant est trivial, et de plus il ne dépend plus de l'assembleur cible.

Le gros intérêt de cette technique est donc de faciliter les compilateurs reciblables : on a la possibilité de déterminer *avant la compilation*, l'automate qui va associer l'ensemble des règles que l'on pourra affecter à chaque nœud. De plus, cet automate est construit automatiquement à partir d'une description formelle de l'assembleur cible : notre ensemble de règles de réécritures.

**Les coûts** Reste à gérer les coûts : pour le moment, on a dans chaque règle de notre étiquette le coût de cette règle. Cela suffit à calculer, en remontant, le coût total du code à ce nœud en fonction des coûts des fils. On note  $R_1@val_1$  pour indiquer que le choix de la règle  $R_1$  pour le nœud donnera un coût total  $val_1$ . Pour notre exemple cela donne :

$$\begin{aligned} Rules(Num_1) &= \{Num@0, Reg \rightarrow Num@1\} \\ Rules(Lab_1) &= \{Lab@0, Num \rightarrow Lab@0, Reg \rightarrow Lab@1\} \\ Rules(+_1) &= \left\{ \begin{array}{l} Reg \rightarrow +(Reg_1, Reg_2)@3, Reg \rightarrow +(Reg_1, Num_2)@2, Reg \rightarrow +(Num_1, Reg_2)@2, \\ R10P2 \rightarrow +(Reg, Reg)@2, R11P2 \rightarrow +(Reg, Num)@1, R12P2 \rightarrow +(Num, Reg)@1, \end{array} \right\} \\ Rules(Ref_1) &= \left\{ \begin{array}{l} Reg \rightarrow Ref(Reg_1)@3, Reg \rightarrow Ref(R10P2)@3, \\ Reg \rightarrow Ref(R11P2)@2, Reg \rightarrow Ref(R12P2)@2 \end{array} \right\} \end{aligned}$$

Avec ce système un nœud contient l'information du coût total du sous arbre pour chaque règle pouvant implémenter le nœud. Ici on a effectué une première sélection en ne conservant que les nœuds fils les moins chers pour un type de stockage en mémoire donné (sachant que l'on n'enlève pas les règles provenant de décomposition de règles complexes du type  $R10P11$ ). Si maintenant dans chaque nœud on ne conserve que les nœuds les moins chers pour un stockage donné, on obtient :

$$\begin{aligned} Rules(Num_1) &= \{Num@0, Reg \rightarrow Num@1\} \\ Rules(Lab_1) &= \{Lab@0, Num \rightarrow Lab@0, Reg \rightarrow Lab@1\} \\ Rules(+_1) &= \left\{ \begin{array}{l} Reg \rightarrow +(Reg_1, Num_2)@2, Reg \rightarrow +(Num_1, Reg_2)@2, \\ R10P2 \rightarrow +(Reg, Reg)@2, R11P2 \rightarrow +(Reg, Num)@1, R12P2 \rightarrow +(Num, Reg)@1, \end{array} \right\} \\ Rules(Ref_1) &= \{ Reg \rightarrow Ref(R11P2)@2, Reg \rightarrow Ref(R12P2)@2 \} \end{aligned}$$

Dans notre exemple, on voit que l'on peut ne garder que deux pavages pour ce nœud  $Ref_1$  : les pavages correspondant à l'application des règles  $\langle 6, 11 \rangle$  et  $\langle 22, 8, 12 \rangle$ . Si l'on choisit une règle pour la racine, par exemple la règle 11, en re-parcourant une dernière fois l'arbre en partant de la racine, on peut supprimer les règles ne devant pas être utilisées, cela donne :

$$\begin{aligned} Rules(Num_1) &= \{Num@0\} \\ Rules(Lab_1) &= \{Reg \rightarrow Lab@1\} \\ Rules(+_1) &= \{ R11P2 \rightarrow +(Reg, Num)@1 \} \\ Rules(Ref_1) &= \{ Reg \rightarrow Ref(R11P2)@3 \} \end{aligned}$$

Cette méthode permet de générer le code en 3 passes sur l'arbre : une passe *bottom up* qui ne conserve que les meilleures règles pour chaque nœud, une passe *top down* qui choisit la règle à utiliser pour chaque nœud en fonction du meilleur coût pour le nœud racine et enfin un nouvelle passe *bottom up* pour générer le code.

Dans cette approche, les étiquettes sont calculées au moment de la compilation. Or on voudrait précalculer les nouvelles étiquettes, en incluant le coût. Mais alors on ne va plus avoir un ensemble fini d'étiquette, les coûts pouvant grossir arbitrairement. Si l'on désire utiliser cette

approche (pour écrire un générateur de générateur de code), on va *normaliser* les étiquettes en stockant pour chaque règle "combien le choix de telle règle rajoute au coût déjà calculé pour les fils". Pour notre exemple, cela ne change rien pour les nœuds  $Num_1$  et  $Lab_1$  mais pour les autres, on obtiendrait :

$$Rules(+_1) = \left\{ \begin{array}{l} Reg \rightarrow +(Reg_1, Num_2)@1, Reg \rightarrow +(Num_1, Reg_2)@1, \\ R10P2 \rightarrow +(Reg, Reg)@1, R11P2 \rightarrow +(Reg, Num)@0, R12P2 \rightarrow +(Num, Reg)@0, \end{array} \right\}$$

$$Rules(Ref_1) = \{ Reg \rightarrow Ref(R11P2)@0, Reg \rightarrow Ref(R12P2)@0 \}$$

On peut faire cela systématiquement pour toutes les possibilités comme on l'a fait précédemment. Un avantage est que cette fois, on ne garde que les règles qui ont le meilleur coût pour un type stockage de résultat donné.

On peut montrer que l'on récupère un ensemble fini d'étiquettes et on peut construire la table de transition que l'on appelle la "table de transition consciente des coûts" (*cost conscious next state table*).

On a donc une technique qui permet de produire un générateur de générateurs de code : en fonction des opérateurs et de l'ensemble de règles proposées, on construit un automate qui travaillera sur un arbre. À la compilation, cet automate annote chaque nœud avec l'état. On peut aussi, calculer au vol le coût total associé à chaque nœud. Ensuite, le processus est le même que précédemment. C'est la technique dite "BURS" (*bottom up rewriting system*).

Plusieurs variantes de cette technique existent, En particulier, on peut linéariser l'AST dans un chaîne en forme préfixée, le problème devient alors un problème de réécriture de chaînes de caractères (*string matching*). On peut utiliser des algorithmes spécifiques pour cela. Cette approche est souvent référencée sous l'appellation "Graham-Glanville" du nom de ses inventeurs.

### 9.3 Sélection d'instruction par fenêtrage

Une autre technique populaire permet de maîtriser la complexité du traitement en n'optimisant que localement. L'idée derrière l'optimisation par fenêtrage est que l'on peut grandement améliorer la qualité du code en examinant un ensemble restreint d'opérations adjacentes. L'optimiseur travaille sur du code assembleur, il a une *fenêtre coulissante* (*sliding windows, peephole*) qui se promène sur le code. À chaque étape, il examine le code présent dans la fenêtre et recherche des schémas prédéfinis (*pattern*) qu'il remplacera par d'autres plus efficaces. La combinaison d'un nombre limité de schémas et d'une fenêtre de petite taille peut aboutir à des traitements très rapides.

Un exemple classique est un store suivi d'un load :

$$\begin{array}{l} storeAI \ r_1 \Rightarrow r_0, 8 \\ loadAI \ r_0, 8 \Rightarrow r_{15} \end{array} \Rightarrow \begin{array}{l} storeAI \ r_1 \Rightarrow r_0, 8 \\ i2i \ r_1 \Rightarrow r_{15} \end{array}$$

D'autres exemples : une identité algébrique simple, un saut suivi d'un saut :

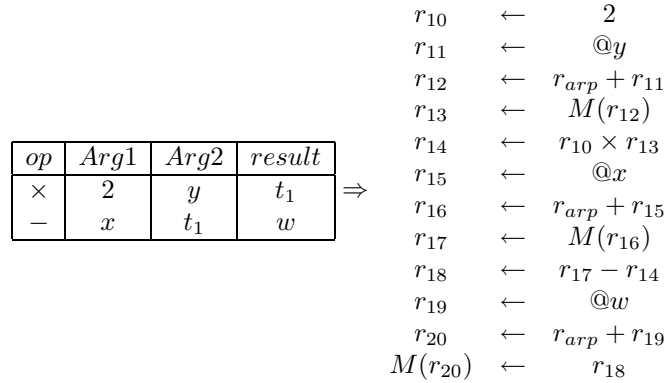
$$\begin{array}{l} addI \ r_2, 0 \Rightarrow r_7 \\ mult \ r_4, r_7 \Rightarrow r_{10} \end{array} \Rightarrow \begin{array}{l} mult \ r_4, r_2 \Rightarrow r_{10} \\ \\ \\ jumpI \ \rightarrow L_{10} \\ L_{10} : jumpI \ \rightarrow L_{11} \end{array} \Rightarrow \begin{array}{l} jumpI \ \rightarrow L_{11} \\ L_{10} : jumpI \ \rightarrow L_{11} \end{array}$$

À l'origine, appliquant des schémas de réécriture très simple, les sélecteurs d'instructions par fenêtrage ont progressivement intégré des techniques de calcul symbolique. Le traitement est maintenant décomposé en trois étapes :

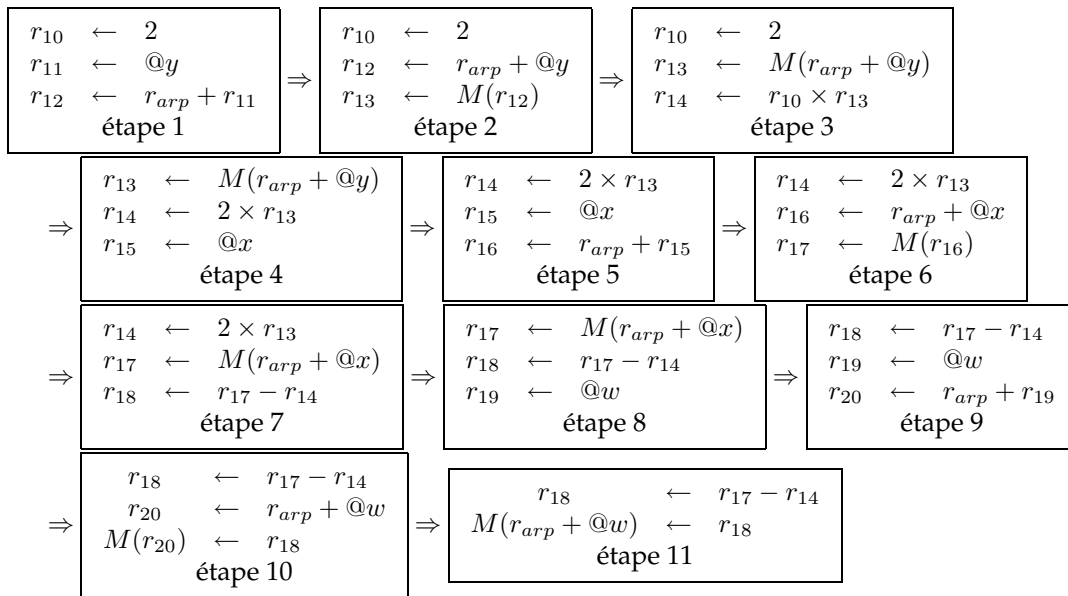
1. Expansion (*expander*), reconnaissance du code et construction d'une représentation interne de bas niveau (*low level IR* : LLIR). Par exemple, les effets de bords comme l'affectation des registres de code condition sont mentionnés explicitement. Cette réécriture peut être faite localement sans connaissance du contexte.

2. Simplification (*simplifier*), application de règles de réécriture en une passe simple sur la LLIR. Les mécanismes les plus utilisés sont la substitution (*forward substitution*), les simplifications algébriques ( $x+0 = x$ ), la propagation de constante ( $12+5 = 17$ ), tout en n'observant qu'une fenêtre limitée du code à chaque étape. C'est la partie clé du processus.
3. Génération (*matcher*), transformation de l'IR en l'assembleur cible.

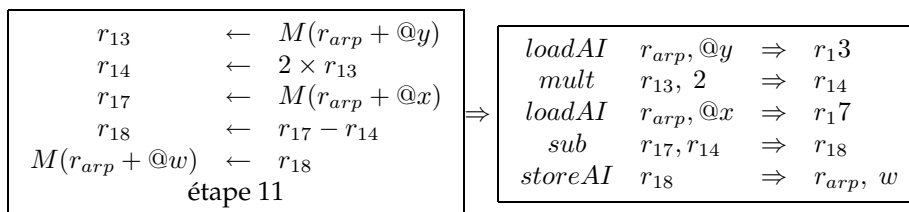
Voici un exemple de transformation complète. Considérons le code suivant correspondant à  $w \leftarrow x - 2 \times y$  :



Supposons une de fenêtre 3 instructions :



Le code restant est donc le suivant et après réécriture :



Notons que pour effectuer certaines transformations (la suppression de certains registres), on a besoin de savoir que les valeurs sont mortes. La reconnaissance des valeurs mortes joue un rôle essentiel dans l'optimisation. Pourtant c'est typiquement une information non-locale. En réalité, l'information utile est recherchée pendant la phase d'expansion, l'optimiseur peut maintenir une

liste de valeurs vivantes par instruction en traduisant le code à rebrousse poil lors de l'expansion (en partant de la fin). Simultanément, il peut détecter les valeurs mortes après leur dernière utilisation (cela est effectivement appliqué uniquement à l'intérieur d'un bloc de base)

La présence d'opérations de contrôle du flot complique la tâche du simplifieur : la méthode la plus simple est d'effacer la fenêtre lorsqu'on entre dans un nouveau bloc de base (cela évite de travailler avec des instructions qui n'auront peut-être pas lieu à l'exécution). On peut aussi essayer de travailler sur le contexte des branchements. L'élimination des labels morts est essentielle, l'optimiseur peut maintenir un compteur de référence des labels et agréger des blocs lorsqu'un label est supprimé, voir supprimer carrément des blocs entiers. Le traitement des opérations prédiquées doit être prévu aussi.

Comme de nombreuses cibles offrent du parallélisme au niveau des instructions, les instructions consécutives correspondent souvent à des calculs indépendants (pour pouvoir faire les calculs en parallèle). Pour améliorer son efficacité, l'optimiseur peut travailler avec une fenêtre virtuelle plutôt qu'une fenêtre physique. Avec une fenêtre virtuelle, l'optimiseur considère un ensemble de calculs qui sont connectés par les flots de valeurs (technique des *échelles*). Pour cela, pendant l'expansion, l'optimiseur peut relier chaque définition avec la prochaine utilisation de la valeur dans le même bloc.

Étendre cette technique au delà du bloc nécessite une analyse plus poussée des utilisations atteintes (*reaching definitions*). Une définition peut atteindre plusieurs premières utilisations et une utilisation peut correspondre à plusieurs définitions.

L'optimisation par fenêtrage a été beaucoup utilisée pour les premières machines, mais avec l'avènement des instructions sur 32 bits, il est devenu très difficile de générer tous les schémas possibles à la main. On a donc proposé des outils qui génèrent le *matcher* à partir d'une description de l'assembleur de la machine cible. De même, de plus en plus de compilateurs génèrent directement la LLIR en maintenant à jour les listes de valeurs mortes.

Le compilateur GCC utilise ces principes. La passe d'optimisation travaille sur une LLIR appelée *register transfer langage*, la phase de génération de code peut cibler le code pour une grande variété d'architectures.

## 10 Allocation de registres

L'allocateur de registres a une influence très importante sur les performances du code généré du fait de l'écart croissant entre la rapidité de la mémoire et des processeurs. Il prend en entrée un code en représentation intermédiaire linéaire avec un nombre de registres virtuels non borné, et produit un programme équivalent

- compatible avec l'ensemble de registres physiques disponibles sur la machine cible
- et minimisant les accès mémoire.

La sémantique de la plupart des langages de programmation étant orientée mémoire, certains accès mémoire seront toutefois obligatoires (en entrée et en sortie de procédure par exemple). À ce sujet, l'utilisation de pointeurs et l'existence de valeurs ambiguës (pouvant être pointées par plusieurs variables) vont aussi obliger à conserver certaines valeurs en mémoire. La suppression de valeurs ambiguës peut donc améliorer significativement les performances du code généré.

Lorsque l'allocateur ne peut pas garder une valeur dans un registre, il insère le code pour stocker cette valeur dans la mémoire et charger cette valeur de la mémoire lorsqu'on l'utilise. Ce processus coûteux est appelé *spill*.

### Remarques

- Le traitement de l'allocateur recouvre en fait deux traitements distincts (généralement faits en une seule passe) : l'*allocation* de registres et l'*assignation* de registres. L'allocation assure simplement qu'à chaque cycle, la machine possédera suffisamment de registres pour exécuter le code. L'assignation de registre assigne effectivement un registre physique de la machine à chaque registres du code sortant de l'allocation de registre.
- Les registres de la machine cible sont classés par *classe de registre*. Quasiment toutes les architectures ont des registres *généraux* (*general purpose registers*) et des registres flottants (*floating point register*). Les registres de code conditions sont aussi spécifiques à chaque architecture. L'IA-64 a une classe de registres spécifiques pour la prédication et pour les adresses cibles de branchement. Enfin, beaucoup de systèmes imposent, pour rendre les appels de fonction moins coûteux, que les premiers paramètres effectifs soient passés dans certains registres réservés. Etc.
- L'allocation de registres est quasiment toujours un problème NP-complet. Le cas simple d'un bloc de base avec une taille uniforme de données et un accès uniforme à la mémoire peut être résolu optimalement en temps polynomial, mais dès que l'on essaye de modéliser une situation un peu plus réaliste (registres dédiés, valeurs devant être stockées en mémoire, plusieurs blocs de base) le problème devient NP-complet. L'assignation de registres est en général un problème polynômial

### 10.1 Variables, registres virtuels, valeurs et durées de vie

Les variables du programme sources ont une sémantique orientée mémoire, mais il sera souvent possible de maintenir leur valeur dans des registres tout en assurant que cette sémantique est conservée. On considérera dans la suite que cette sémantique est assurée par des opérations explicites de lecture et d'écriture mémoire dans l'IR en entrée de l'allocateur. Ainsi, désormais, les variables du programme source ne sont pas traitées différemment des variables intermédiaires ajoutées par le compilateur.

Toutes ces variables (celles du programme source, et les variables intermédiaires) sont exprimées dans l'IR en entrée de l'allocateur par des *registres virtuels*. On dira toutefois *variable* pour faire plus court.

Cela dit, ce ne sont pas exactement ces variables qu'il va falloir assigner à des registres physiques : en effet, une variable peut prendre dans le programme des *valeurs* différentes, et il sera possible que l'allocation optimale conserve l'une de ces valeurs dans un registre, mais *spille* une autre.

La notion importante est celle de *durée de vie*. Dans un bloc de base, la définition d'une durée de vie est assez intuitive : il s'agit de l'ensemble des nœuds entre une définition de la valeur d'une variable et sa dernière utilisation avant d'être redéfinie. On voit qu'une variable aura autant de



durées de vie que de valeurs redéfinies. Considérons par exemple le code suivant :

```

1.  loadI    @stack  =>  rarp
2.  loadAI   rarp,0   =>  rw
3.  loadI    2        =>  r2
4.  loadAI   rarp,8   =>  rx
5.  loadAI   rarp,16  =>  ry
6.  loadAI   rarp,24  =>  rz
7.  mult     rw,r2    =>  rw
8.  mult     rw,rx    =>  rw
9.  mult     rw,ry    =>  rw
10. mult     rw,rz    =>  rw
11. storeAI  rw       =>  rarp,0

```

La valeur de  $rarp$  est définie une fois au début du bloc, ensuite elle est utilisée jusqu'à la fin du bloc. Sa durée de vie est l'ensemble de nœuds  $[1, 11]$ . Considérons  $rw$ , elle est définie par les opérations 2,7,8,9,10 et utilisée par les opérations 7 à 11, elle définit donc les durées de vie :  $[2, 7]$ ,  $[7, 8]$ ,  $[8, 9]$ ,  $[9, 10]$  et  $[10, 11]$ . Le bloc définit aussi les durées de vie suivantes :  $[3, 7]$  pour  $r2$ ,  $[4, 8]$  pour  $rx$ ,  $[5, 9]$  pour  $ry$ ,  $[6, 10]$  pour  $rz$ .

Sur un CFG avec des branchements, cette notion est plus complexe. Une durée de vie est un ensemble connexe de nœuds du CFG associé à une valeur d'une variable. Deux nœuds  $A$  et  $B$  sont dans la même durée de vie associée à la variable  $v$  s'il existe un chemin  $P$  dans le CFG d'une définition de  $v$  à une utilisation de  $v$  sans redéfinition de  $v$ , tel qu'il existe un chemin de  $A$  à  $B$  qui est un sous-chemin de  $P$ .

Et donc, ce sont de telles durées de vies qu'on tâchera d'allouer à des registres physiques.

## 10.2 Vivacité

Pour cela, il est plus simple de changer de point de vue : au lieu de considérer le CFG tout entier, plaçons nous sur une arête, et considérons l'ensemble des variables qui y sont vivantes, c'est-à-dire qui ont une durée de vie contenant cette arête.

La définition de la vivacité est alors moins vaseuse car on ne considère plus qu'une valeur par variable, et l'on identifie les deux : sur une arête  $e$  du CFG, une variable  $v$  est vivante si

- il existe un chemin partant du début de la procédure passant par  $e$  sur lequel la valeur de  $v$  a été définie
- et il existe un chemin de  $e$  à une utilisation de  $v$  sur lequel  $v$  n'est pas redéfinie.

Insistons sur le fait qu'on parle de la vivacité d'une variable *sur une arête*, pas en un nœud du CFG. Les vivacités en sortie et en entrée d'un nœud sont en général différentes, et c'est même comme cela qu'on les calcule : Pour tout nœud  $n$  du CFG, on définit

- $Def(n)$  l'ensemble des variables définies par le nœud  $n$ . Pour une affectation, ce sera un singleton, parfois un couple si l'on tient compte des effets sur les drapeaux ou sur des registres de prédicats éventuels. Si  $n$  est un appel de procédure,  $Def(n)$  peut être arbitrairement gros.
- $Use(n)$  l'ensemble des variables utilisées par ce nœud.
- $LiveIn(n)$  l'ensemble des variables vivantes en entrée de  $n$ . Si  $n$  a plusieurs arêtes entrantes, c'est l'union sur toutes les arêtes entrantes des ensembles de variables vivantes sur ces arêtes. Autrement dit,  $v \in LiveIn(n)$  ssi il existe une arête  $e$  entrant dans  $n$  telle que  $v$  est vivante sur  $e$ .
- $LiveOut(n)$  l'ensemble des variables vivantes en sortie de  $n$  (à nouveau, c'est l'union sur toutes les arêtes sortantes des ensembles de variables vivantes sur ces arêtes)

Une fois ces définitions bien digérées, il est facile de les relier entre elles : une variable  $v$  est vivante à l'entrée d'un nœud  $n$  si elle est utilisée dans  $n$  ( $v \in Use(n)$ ), ou si elle est vivante en sortie de  $n$  sans avoir été définie dans le  $n$  ( $v \in LiveOut(n) \setminus Def(n)$ ).

On obtient ainsi les *équations data-flow* suivantes :

$$LiveOut(n) = \bigcup_{s \in succ(n)} LiveIn(s) \quad (1)$$

$$LiveIn(n) = Use(n) \cup (LiveOut(n) \setminus Def(n)) \quad (2)$$

Ici, l'ensemble *LiveOut* d'un nœud est une fonction des ensembles *LiveIn* de ses successeurs : l'information remonte le long des arcs du graphe de flot de contrôle. On verra pour l'optimisation d'autres équations *data-flow* où l'information descend.

Ce type d'équations permet de calculer les ensembles itérativement par un algorithme de point fixe, après avoir calculé facilement les informations locales à chaque nœud (les ensembles *Def* et *Use*). Pour montrer la terminaison de l'algorithme, il faut vérifier que tous les ensembles sont finis et grossissent (l'algorithme s'arrête lorsqu'ils ne grossissent plus, en tout cas). Pour montrer qu'il est correct, il suffit de montrer que l'information construite a une structure de treillis.

### 10.3 Allocation globale

**Construction des durées de vie** On doit d'abord calculer les durées de vies, qui sont (rappel) des ensembles connexes de nœuds du CFG associés à une variable.

À ce point il reste encore une certaine confusion entre une variable/un registre virtuel, et une valeur de cette variable. Supposons donc qu'on part d'un CFG en forme SSA, ce qui supprimera cette confusion en identifiant les deux notions. Mais pourquoi a-t-on calculé la vivacité, alors ? Eh bien, pour obtenir la forme SSA optimisée (*pruned SSA*) : cf le TD.

Les fonctions  $\phi$  rendent explicite le fait que des définitions distinctes sur des chemins distincts atteignent une même référence. Deux définitions atteignant la même fonction  $\phi$  définissent la même durée de vie car la fonction *phi* crée un nom représentant les deux valeurs.

Pour construire les durées de vie à partir de la forme SSA, l'allocateur commence donc par assigner à chaque nom un ensemble de nœud, qui en SSA est un intervalle. Ensuite, il examine les fonctions  $\phi$  du programme et fait l'union des ensembles pour tous les paramètres d'une même fonction  $\phi$ . Lorsque toutes les fonctions  $\phi$  ont été traitées, les ensembles résultant représentent les durées de vie maximales du programme. Au passage, cette opération sort de la forme SSA.

**Évaluation du coût d'un spill** Il reste à évaluer les coûts des spills éventuels. Cette évaluation est délicate car elle peut dépendre des exécutions. Par exemple, le coût total d'un spill dans une boucle dépend du nombre de tours dans la boucle, qui est souvent inconnu à la compilation.

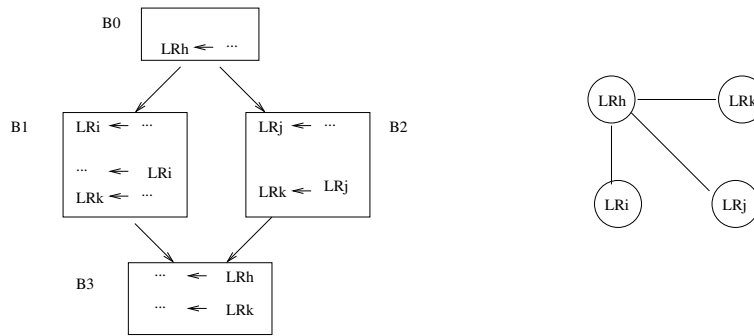
En général les spills sont envoyés sur la pile, ce qui évite d'utiliser d'autres registres pour le calcul de l'adresse. Le compilateur peut alors évaluer, pour chaque durée de vie, le coût du spill de la variable. Certaines durée de vie peuvent avoir un coût négatif. Par exemple, un store suivi d'un load avec aucune utilisation entre temps devra systématiquement être spillé. D'autres durées de vie auront un coût infini. Par exemple, pour une utilisation qui suit immédiatement une définition, un spill ne réduira pas le nombre de registres utilisés. Une durée de vie aura un coût infini si aucune durée de vie qui interfère ne finit entre la définition et l'utilisation et si moins de  $k - 1$  valeurs sont définies entre les définitions et les utilisations (il faut éviter la situation où  $k$  durées de vie qui n'interfèrent pas ont toutes un coût infini).

Enfin, pour tenir compte de la fréquence d'utilisation des différents blocs la plupart des compilateurs utilisent des heuristiques assez primaires : chaque boucle s'exécute 10 fois (donc une double boucle 100 fois), les branchements sont pris une fois sur deux, etc. Le coût du spill d'une référence est alors :

$$\text{coût des opérations mémoire ajoutées} \times \text{fréquence d'exécution}$$

**Graphe d'interférence** Les interférences représentent la compétition entre les valeurs pour les noms des registres physiques. Considérons deux variables dont la durées de vie :  $LR_i$  et  $LR_j$  (dans la suite on identifie les variables avec leur durée de vie). Si il existe une opération où les deux variables  $LR_i$  et  $LR_j$  sont vivantes, alors ces deux variables ne peuvent pas utiliser le même registre physique. On dit alors que  $LR_i$  et  $LR_j$  *interfèrent*.

Le compilateur va construire un graphe d'interférence  $I$ . Les nœuds de  $I$  représentent les durées de vie (ou les variables). Les arcs non orientés représentent les interférences. Il existe un arc  $(n_i, n_j)$  si et seulement si il existe une arête sur laquelle les variables correspondantes sont toute deux vivantes en même temps. Considérons l'exemple suivant :



Si le compilateur peut construire une  $k$ -coloration du graphe d'interférence, c'est à dire colorer chaque nœud d'une couleur différente de tous ses voisins, avec uniquement  $k$  couleurs, alors il peut associer une couleur à chaque registre physique et produire une allocation légale. Pour notre exemple, si l'on associe une couleur  $LR_h$ , les trois autres variables peuvent recevoir la même couleur, différente de celle de  $LR_h$ , le graphe est 2-colorable.

Le problème de déterminer si un graphe est  $k$ -colorable est NP-complet, les compilateurs utilisent donc des heuristiques rapides qui ne trouvent pas toujours la solution lorsqu'elle existe. Ensuite, le compilateur a besoin d'une stratégie à exécuter lorsque l'on ne peut pas colorer un nœud. Il peut soit *spill* la variable soit *séparer* la durée de vie. *Spiller* signifie que toutes les références à la variable sont placées en mémoire et la durée de vie est séparée en une multitude de minuscules durées de vie. *Séparer* la durée de vie signifie que l'on crée des durées de vie plus petites mais non triviales, en insérant, par exemple des copies ou des spill à certains endroits. Dans les deux cas les graphes d'interférences résultant sont différents, si la transformation est effectuée intelligemment, le nouveau graphe est plus facile à colorer.

**Coloration top down** Cette heuristique consiste à partitionner le graphe en deux types de nœuds : les nœuds non contraints qui ont moins de  $k - 1$  voisins et les nœuds contraints qui ont au moins  $k$  voisins. Quelles que soient les couleurs des nœuds contraints, les nœuds non contraints pourront toujours être colorés. Les premiers nœuds contraints à être colorés doivent être ceux dont les coûts estimés de spill sont les plus importants. Cet allocateur alloue les couleurs suivant cet ordre de priorité. Lorsqu'il rencontre un nœud qu'il ne peut pas colorer, il sépare ou spill le nœud (pas de backtrack), en espérant que ce nœud sera plus facile à spill que les nœuds déjà alloués. Séparer le nœud peut se révéler intéressant car il peut créer des durée de vie non contraintes.

**Coloration bottom up** La seconde stratégie utilise les mêmes principes de base : on cherche les durées de vie, on construit un graphe d'interférence, on essaye de le colorer et on insère des spill lorsque l'on ne peut pas. La différence vient de l'ordre dans lequel on considère les variables. L'algorithme utilisé est le suivant :

```

Initialiser la pile
Tant que ( $N \neq \emptyset$ )
  si  $\exists n \in N$  avec  $deg(n) < k$ 
     $n_{\text{nœud}} \leftarrow n$ 
  sinon
     $n_{\text{nœud}} \leftarrow n$  choisi dans  $N$ 
  enlever  $n_{\text{nœud}}$  et ses arcs de  $I$ 
  empiler  $n_{\text{nœud}}$  sur la pile.

```

L'allocateur détruit le graphe dans un certain ordre, Le point important ici c'est qu'il empile les nœuds non contraints avant les nœuds contraints et le fait d'enlever un nœud contraint dans un graphe qui n'a plus de nœud non contraints peut faire apparaître des nœud non contraint. L'algorithme reconstruit ensuite le graphe d'interférence dans l'ordre de la pile :

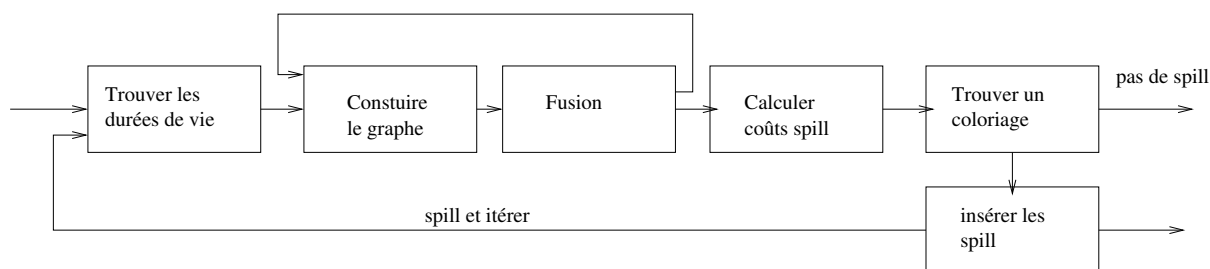
Tant que ( $pile \neq \emptyset$ )  
 $nœud \leftarrow sommet\ de\ pile$   
insérer  $nœud$  et ses arcs dans  $I$   
colorer le nœud

Si un nœud ne peut pas être coloré, il est laissé incolore. Lorsque le graphe a été reconstruit, si tous les nœuds sont colorés, l'allocation est terminée. Si il reste des nœuds incolores, l'allocateur sépare ou spill. Le code est alors réécrit et le processus complet recommence (en général deux ou trois itérations en tout).

Le succès de l'algorithme vient du fait qu'il choisit les nœuds dans un ordre "intelligent". Pour cela, lorsque tous les nœuds du graphe sont contraints, il utilise la *spill metric* : il prend le nœud qui minimise le rapport *coût estimé du nœud / degré courant du nœud*.

**Amélioration** Lorsque deux durées de vie  $LR_i$  et  $LR_j$  qui n'interfèrent pas sont reliées par une copie de registre à registre :  $i2i\ LR_i \Rightarrow LR_j$ , on peut fusionner les deux durées de vie, supprimer l'opération de copie et remplacer partout  $LR_j$  par  $LR_i$ . Cela ne peut que faire décroître le degré des nœuds du graphe (en particulier ceux qui interféraient avec  $LR_i$  et  $LR_j$ ).

L'ordre dans lequel on effectue ces fusion (*coalescing*) est important car il peut arriver que deux fusions soient possibles mais incompatibles. En pratique, les fusions sont effectuées d'abord pour les boucles les plus imbriquées. Le schéma global d'un allocateur de registre est donc :



On peut généraliser le graphe d'interférence et ajouter les contraintes sur les types de registres (par exemple que les valeurs flottantes ne peuvent pas aller dans les registres entiers). Pour cela, il suffit de rajouter un nœud au graphe par registre réel et de mettre un arc entre une variable et un registre physique si la variable ne peut pas être stockée dans le registre.

## 10.4 Allocation et sélection d'instruction dans quel ordre ?

Dans quel ordre faut-il réaliser allocation de registres et sélection d'instruction ? En principe, l'allocation de registre est une des dernières étapes de la compilation, mais on a vu que la sélection d'instruction peut réduire le nombre de variables présentes dans le code en fusionnant des instructions (ce qui économise des variables intermédiaires). Faire d'abord la sélection d'instruction va donc faciliter l'allocation. Oui mais la sélection d'instruction telle qu'on l'a vue suppose toutes les variables en mémoire, si après coup on en met en registres l'optimalité n'est plus garantie. Damned.

En pratique on réalisera plutôt l'allocation avant la sélection, et en cas de libération d'un registre dans un endroit contraint, on pourra refaire une passe d'allocation après la sélection.

Le même problème se pose avec la plupart des techniques d'optimisations : il est en général très difficile de garantir que l'optimisation d'un critère ne dégrade pas un autre critère optimisé au cours d'une étape précédente. En pratique, on bricole encore beaucoup à ce niveau.

## 10.5 Algorithmes d'allocation de registres locale (intérêt historique)

Lorsque l'on ne s'intéresse qu'à un bloc de base, l'allocation de registres est *locale*. Le bloc consiste en une suite de  $N$  opérations  $op_1, \dots, op_N$  de la forme  $op_i\ vr_{i1}, vr_{i2} \rightarrow vr_{i3}$  ( $vr$  pour

*virtual register*). Le nombre total de registres virtuels utilisés dans le bloc est  $MaxVR$ , le nombre total de registres disponibles sur la machine est  $k$ . On suppose pour l'instant qu'aucune contrainte ne vient des autres blocs du programme. Si  $MaxVR > k$ , l'allocateur doit sélectionner des registres à spiller. On va voir deux approches pour résoudre ce problème.

La première méthode est d'utiliser la fréquence d'utilisation comme heuristique pour le choix des registres à spiller (*top down approach*). Plus un registre sera utilisé dans le bloc, moins il aura de chance d'être spillé. Lors du compte des registres, il faut prévoir qu'un petit nombre de registres en plus seront nécessaires pour traiter les opérations sur les valeurs stockées en mémoire après les spill : chargement des opérandes, calcul et stockage du résultat (typiquement entre deux et quatre registres, on désigne ce nombre par  $nbRegSup$ ). Un algorithme possible pour ce traitement est le suivant :

1. Calculer un score pour classer les registres virtuels en comptant toutes les utilisations de chaque registre (passe linéaire sur le bloc).
2. trier les registres par score.
3. Assigner les registres par priorité : les  $k - nbRegSup$  registres sont assignés à des registres physiques
4. Réécrire le code : Lors d'une seconde passe sur le code, chaque registre assigné à un registre physique est remplacé par le nom du registre physique. Chaque registre virtuel qui n'est pas assigné utilise les registres conservés à cet effet, la valeur est chargée avant chaque utilisation et stockée après chaque définition.

La force de cette approche c'est qu'elle conserve les registres très fréquemment utilisés dans des registres physique. Sa faiblesse, c'est qu'elle dédie un registre physique à un registre virtuel sur toute la durée du bloc, alors qu'il peut y avoir un très fort taux d'utilisation d'un registre virtuel au début du bloc et plus d'utilisation après.

Une autre approche part de considérations bas niveau (*bottom-up approach*). L'idée est de considérer chaque opération et de remettre en question l'allocation de registres à chaque opération. L'algorithme global est le suivant :

Pour chaque opération  $i$  de 1 à  $N$ , de la forme  $op_i \ vr_{i1}, vr_{i2} \rightarrow vr_{i3}$

```

 $pr_x \leftarrow ensure(vr_{i1}, class(vr_{i1}))$ 
 $pr_y \leftarrow ensure(vr_{i2}, class(vr_{i2}))$ 
si  $vr_{i1}$  n'est plus utilisé après  $i$  alors
     $free(pr_x, class(pr_x))$ 
si  $vr_{i2}$  n'est plus utilisé après  $i$  alors
     $free(pr_y, class(pr_y))$ 
 $r_z \leftarrow ensure(vr_{i3}, class(vr_{i3}))$ 
emit  $op_i \ r_x, r_y \Rightarrow r_z$ 

```

Le fait de vérifier si un registre n'est pas utilisé permet d'éviter de spiller un registre alors que l'opération libère un registre. La procédure *ensure* doit, pour chaque registre virtuel rencontré, si nécessaire charger une valeur depuis la mémoire dans un registre, et dans tous les cas retourner un nom de registre physique utilisé. Son pseudo-code est le suivant :

```

ensure( $vr, class$ )
si  $vr$  est déjà assigné au registre physique  $pr$ 
     $result \leftarrow pr$ 
    mettre à jour  $class.Next[pr]$ 
sinon
     $result \leftarrow allocate(vr, class)$ 
    émettre le code pour charger  $vr$  dans  $result$ 
return  $result$ 

```

Si l'on oublie pour l'instant la mise à jour de  $class.Next$  qui est détaillée un peu plus loin, l'algorithme *ensure* est très intuitif. L'intelligence de l'algorithme d'allocation est dans les procédures

*allocate* et *free*. Pour les décrire, nous avons besoin d'une description plus précise de ce que l'on va appeler une *classe*. Dans l'implémentation, la classe va connaître l'état exact de chacun des registres physiques de cette classe : est-il alloué à un registre virtuel, si oui lequel et l'index de la prochaine référence à ce registre virtuel (le fameux *class.next[pr]*). Cette dernière valeur permettra, lorsque l'on devra spiller un registre, de choisir celui qui sera rechargé de la mémoire le plus tard possible. La classe contient aussi une structure (par exemple une pile) contenant tous les registres physiques non utilisés. Voici un moyen de réaliser une telle structure de donnée en C et l'initialisation correspondante :

```

struct class {
    int Size;
    int Name[Size]; /* Nom du registre virtuel */
    int Next[Size]; /* Distance à la prochaine utilisation */
    int Free[Size]; /* Booleen */
    int Stack[Size]; /* Pile de registres physiques libres */
    int StackTop;
}

```

```

initialize(class, size)
    class.Size ← size;
    class.StackTop ← -1
    pour i de 0 à size - 1
        class.Name[i] ← ⊥
        class.Next[i] ← ∞
        class.Free[i] ← true
    push(i, class)

```

On peut maintenant précisément décrire le comportement des procédures *allocate* et *free* :

```

allocate(vr, class)
    si (class.StackTop ≥ 0)
        pr ← pop(class)
    sinon
        pr ← registre qui minimise class.Next[pr]
        stocker le contenu de pr
    class.Name[pr] ← vr
    class.Next[pr] ← dist(vr)
    class.Free[pr] ← false

```

```

free(pr, class)
    si (class.Free[pr] ≠ true) alors
        push(pr, class)
        class.Name[pr] ← ⊥
        class.Next[pr] ← ∞
        class.Free[pr] ← true

```

La procédure *allocate* renvoi un registre physique libre (i.e. non alloué) de la classe si il en existe un. Sinon, elle sélectionne, parmi les registres physiques de la classe, celui qui contient une valeur qui est utilisée le plus loin possible dans le futur. Elle stocke cette valeur en mémoire et alloue le registre physique pour *vr*. La fonction *dist(vr)* renvoi l'index dans le bloc courant de la prochaine référence à *vr*. Le compilateur peut annoter chaque référence avec les valeurs de la fonction *dist* en faisant une simple passe remontante sur le bloc. La procédure *free* fait les actions nécessaires pour indiquer que le registre est libre (en particulier, elle empile le registre physique sur la pile de sa classe).

Cet algorithme a été proposé par Sheldon Best dans les années 50 pour le premier compilateur Fortran, Il produit des allocations excellentes. Certain auteurs ont prétendus que les allocations étaient optimales, mais il existe des situations pour lesquelles d'autres phénomènes doivent être pris en compte. Par exemple, si une valeur a été obtenue par un *load* et n'a pas été modifiée. Si on doit la spiller on n'a pas besoin de générer un store : la valeur est déjà en mémoire. Une telle valeur (qui ne nécessite pas une recopie en mémoire) est dite *propre*, alors qu'une valeur nécessitant une recopie est dite *sale*.

Considérons une machine avec deux registres et supposons que les valeurs  $x_1$  et  $x_2$  sont déjà dans les registres,  $x_1$  étant propre et  $x_2$  étant sale. Supposons que l'on rencontre les trois valeurs dans cette ordre :  $x_3$   $x_1$   $x_2$ . L'allocateur doit spiller une des valeurs  $x_1$  ou  $x_2$  pour charger  $x_3$ . Comme la prochaine utilisation de  $x_2$  est la plus lointaine, l'algorithme de Best va la spiller et produire la séquence suivante :

```

store x2
load x3
load x2

```

Or si l'on avait choisi de spiller la valeur propre  $x_1$  on obtiendrait le code :

```
load x3
load x1
```

Ce scénario suggère de spiller les valeurs propres d'abord, mais on peut trouver des cas où ce n'est pas vrai : considérons la séquence d'accès  $x_3 x_1 x_3 x_1 x_2$  avec les mêmes conditions initiales. En spillant systématiquement les valeurs propres on obtient le code suivant :

```
load x3
load x1
load x3
load x1
```

Alors qu'en spillant  $x_2$  on utilisera un accès à la mémoire de moins :

```
store x2
load x3
load x2
```

Le fait de prendre en compte la distinction entre valeurs propres et valeurs sale rend le problème NP-complet.

De manière générale, ces allocateurs produisent des résultats bien meilleurs que les allocateurs *top down* car ils allouent un registre physique à un registre virtuel uniquement pour la distance entre deux références au registre virtuel et reconsidèrent cette décision à chaque appel de la fonction *allocate*

## 11 Ordonnancement d'instructions

### 11.1 Exemple introductif

En introduction, on avait utilisé cette portion de code Iloc qui utilisait peu de registres. Ici on a rajouté les dates d'exécution de chaque instruction (en nombre de cycles), en supposant : qu'un chargement depuis la mémoire coûtait trois cycles et qu'une multiplication coûtait deux cycles. On a aussi supposé que les opérateurs étaient pipelinés, c'est à dire que l'on pouvait commencer un nouveau chargement alors que le chargement précédent n'était pas terminé.

```
1    loadAI    r_arp, @w    => r1    // load w
4    add       r1, r1      => r1    // r1 ← w × 2
5    loadAI    r_arp, @x    => r2    // load x
8    mult      r1, r2      => r1    // r1 ← (w × 2) × x
9    loadAI    r_arp, @y    => r2    // load y
12   mult      r1, r2      => r1    // r_w ← (w × 2) × x × y
13   loadAI    r_arp, @z    => r2    // load z
16   mult      r1, r2      => r1    // r1 ← (w × 2) × x × y × z
18   storeAI   r1         => r_arp, @w // écriture de w
```

L'ordonnancement des instructions ci-dessous est meilleur car il peut s'exécuter en 11 cycles au lieu de 18 (mais il utilise un registre de plus).

```
1    loadAI    r_arp, @w    => r1    // load w
2    loadAI    r_arp, @x    => r2    // load x
3    loadAI    r_arp, @y    => r3    // load y
4    add       r1, r1      => r1    // r1 ← w × 2
5    mult      r1, r2      => r1    // r1 ← (w × 2) × x
6    loadAI    r_arp, @z    => r2    // load z
7    mult      r1, r3      => r1    // r_w ← (w × 2) × x × y
9    mult      r1, r2      => r1    // r1 ← (w × 2) × x × y × z
11   storeAI   r1         => r_arp, @w // écriture de w
```

L'ordonnancement de code essaye de cacher les latences des opérations, comme les chargements en mémoire ou les branchements, pour cela il exploite le *parallélisme de niveau instruction* (*instruction level parallelism*, ILP) qui est un parallélisme de grain très fin par opposition au parallélisme de tâche ou au parallélisme entre itérations de boucles.

L'ordonnancement d'instructions prend comme entrée une suite partiellement ordonnée d'instructions, et produit comme sortie une liste des mêmes instructions. Le travail d'ordonnancement d'instructions est extrêmement lié à celui de l'allocation de registres, mais il est généralement admis que résoudre les deux problèmes simultanément est trop complexe. En général l'ordonnancement passe avant l'allocation de registre, mais de plus en plus il est repassé après l'allocation de registre pour améliorer le code.

### 11.2 Matériel contre logiciel

Le débat fait toujours rage sur la question suivante : l'ordonnancement doit-il être réalisé en matériel, ou bien en logiciel par le compilateur ? Les processeurs récents, en effet, ont (presque) tous la capacité d'exécuter les instructions qu'ils reçoivent *dans le désordre*.

**Parenthèse architecturale** Pour ce faire, les instructions arrivant dans le processeur sont placées dans une queue et (pour les jeux d'instructions pauvres en registres comme IA32/x86) leurs registres renommés : le processeur dispose de plus de "vrais" registres (registres architecturaux) que ceux décrits par le jeu d'instruction. Il maintient dans une table la correspondance entre registre du code et registre architectural. De plus, cette table contient les informations de dépendance : si une instruction utilise un registre produit par une autre instruction, la table contient un pointeur soit vers cette instruction dans la queue, soit (si l'instruction productrice a déjà été lancée) un pointeur vers l'unité de calcul qui produira le résultat.



Le matériel a donc à gérer les mises à jour de cette table au fur et à mesure que des instructions sont lancées et se terminent. Il doit aussi, à chaque cycle, choisir dans la queue des instructions qu'il peut lancer en fonction 1/ des dépendances et 2/ des unités de calcul disponibles. Les instructions peuvent donc être lancées dans le désordre. Comme elles ont des latences variables, elles se terminent encore plus dans le désordre.

Le coût en matériel de tout ceci est aggravé de ce que les processeurs récents ("superscalaires") essaient de lancer plusieurs instructions en même temps.

Cette approche a été (et est toujours) poussée par l'héritage des processeurs IA32. En réaction à la complexité mise en jeu, les jeux d'instructions plus récents ont tenté de déléguer au compilateur une part de cette gestion :

- jeu d'instruction explicitement parallèle : une instruction assembleur est un paquet de 4 ou 8 instructions à exécuter en parallèle (approche VLIW pour *very large instruction word*)
- gestion par le compilateur de l'ordonnancement.

Je le redis, le débat fait rage. Les pour et les contre sont les suivants :

- Le matériel est plus coûteux à tout point de vue. Tout silicium consacré à l'ordonnancement produirait peut-être une accélération supérieure s'il était consacré à augmenter le cache, par exemple. De plus, c'est conceptuellement plus simple d'augmenter le cache que d'améliorer l'ordonnanceur matériel. Un bug dans le compilateur est facile à corriger, un bug dans la puce signifie un retard de plusieurs mois qui a souvent été fatal.
- Le logiciel peut considérer une fenêtre d'instructions bien plus grande que le matériel, et ainsi optimiser mieux.
- Mais le matériel a accès à une information dynamique qui est plus précise : il optimise l'ordonnancement du flot de contrôle effectif, alors que le logiciel doit optimiser l'ordonnancement sur tous les flots de contrôle possibles.
- L'approche logicielle s'adapte mal à des générations successives de processeurs, et/ou à des processeurs de marques différentes (Intel et AMD par exemple). Il faut tout recompiler pour obtenir un ordonnancement optimal sur une version donnée d'un processeur.

En pratique les deux extrêmes sont :

- Tout matériel pour les x86. Toutefois le compilateur fait de l'ordonnancement dans les grandes lignes, par exemple cela ne fait jamais de mal de remonter les accès mémoire le plus en avant possible car ils prennent du temps, quelle que soit l'architecture
- Tout logiciel pour l'embarqué, où les aspects de consommation plaident en faveur de moins de matériel, alors que ce n'est pas un problème de tout devoir recompiler.

L'IA64 (Itanium) est une tentative de jeu d'instruction qui explicite le parallélisme mais conserve une partie d'ordonnancement dynamique. Il n'est pas certain qu'on obtient ainsi les avantages des deux approches, par contre il est clair qu'on en conserve les inconvénients...

### 11.3 Le problème de l'ordonnancement

L'ordonnanceur travaille sur le graphe de précedence  $\mathcal{P} = (N, E)$  qui est un graphe de dépendance entre instructions. Chaque noeud  $n \in N$  est une instruction du code original, un arc  $e = (n_1, n_2) \in E$  si et seulement si  $n_2$  utilise le résultat de  $n_1$  comme argument. Chaque noeud a en plus deux attributs : un type d'opération et une durée (ou latence). Le noeud  $n$  s'exécutera sur l'unité fonctionnelle  $type(n)$  et son exécution durera  $delai(n)$  cycles. Les noeuds sans prédécesseurs sont appelés feuilles (il peuvent être ordonnancés n'importe quand). Les noeuds sans successeurs dans le graphe sont appelés racines, ce sont les noeuds les plus contraints.

Étant donné un graphe de précedence, un ordonnancement associe chaque noeud  $n$  à un entier positif  $S(n)$  qui représente le cycle dans lequel il sera exécuté. Il doit respecter les contraintes suivantes :

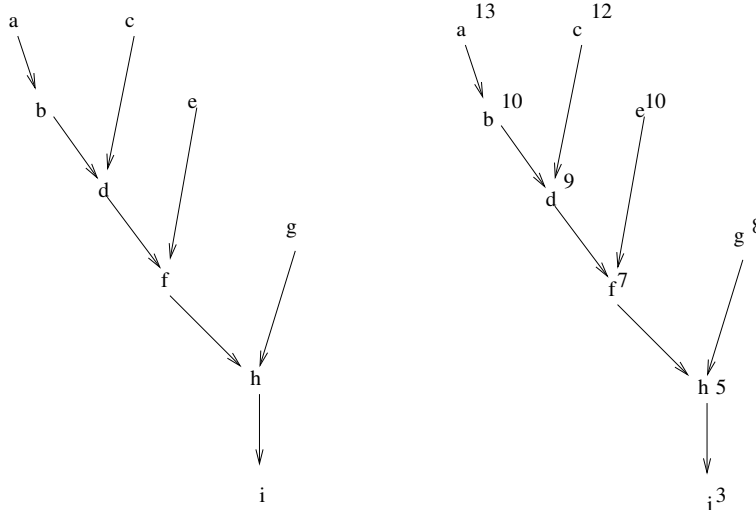
1.  $S(n) \geq 0$  (on suppose aussi qu'il y a au moins une opération telle que  $S(n) = 0$ ).
2. si  $(n_1, n_2) \in E$ ,  $S(n_1) + delai(n_1) \leq S(n_2)$ . Cette contrainte assure la validité de l'ordonnancement.
3. L'ensemble des opérations s'exécutant à un cycle donné ne contient pas plus d'opérations de type  $t$  que la machine cible ne peut en exécuter en parallèle.

On définit alors la longueur ou latence de l'ordonnancement comme

$$L(S) = \max_{n \in N} (S(n) + \text{delay}(n))$$

Le graphe de précédence fait apparaître une quantité intéressante : les *chemins critiques*. L'accumulation des délais en remontant le long des chemins dans le graphe de précédence permet d'annoter chaque noeud avec le temps total minimal qu'il faudrait pour atteindre la racine (c'est à dire en ajoutant tous les délais des opérations sur le plus long chemin jusqu'à une racine). Ci dessous nous avons représenté le graphe de précédence du code original, en annotant chaque noeud avec ce temps. On voit que sur ce graphe, le chemin critique est le chemin *abdfhi*.

a	loadAI	$r_{arp}, @w$	$\Rightarrow r_1$	// load w
b	add	$r_1, r_1$	$\Rightarrow r_1$	// $r_1 \leftarrow w \times 2$
c	loadAI	$r_{arp}, @x$	$\Rightarrow r_2$	// load x
d	mult	$r_1, r_2$	$\Rightarrow r_1$	// $r_1 \leftarrow (w \times 2) \times x$
e	loadAI	$r_{arp}, @y$	$\Rightarrow r_2$	// load y
f	mult	$r_1, r_2$	$\Rightarrow r_1$	// $r_w \leftarrow (w \times 2) \times x \times y$
g	loadAI	$r_{arp}, @z$	$\Rightarrow r_2$	// load z
h	mult	$r_1, r_2$	$\Rightarrow r_1$	// $r_1 \leftarrow (w \times 2) \times x \times y \times z$
i	storeAI	$r_1$	$\Rightarrow r_{arp}, @w$	// écriture de w



On sent bien que l'ordonnancement a intérêt à ordonner l'instruction *a* en premier. Le chemin critique devient alors *cdefhi* qui suggère d'ordonner *c*. Ensuite, *b* et *e* sont à égalité mais *b* a besoin du résultat de *a* qui ne sera disponible que dans un cycle. En continuant ainsi on arrive à l'ordonnement *acebdgphi* qui est celui que l'on cherchait. Cependant, le compilateur ne peut pas réécrire le code avec cette ordonnance en utilisant exactement les mêmes instructions. En effet, *c* et *e* définissent toutes les deux  $r_2$ , si l'on ramène *e* avant *d*, il faut renommer le résultat de *e*. Cette contrainte ne vient pas des dépendances de flot mais par des contraintes de nommage sur les valeurs. On parle d'*anti-dépendance* : une instruction écrit à un endroit qui est lu auparavant par une autre instruction. Il peut aussi y avoir des *dépendances de sortie* lorsque deux instructions écrivent dans la même variable. Un code en SSA ne possède ni anti-dépendance si dépendance de sortie, il ne possède que des *vrai dépendances* ou dépendances de données.

L'ordonnement peut prendre cela en compte de deux manières : soit il respecte les anti-dépendances comme les dépendances du graphe de précédence et n'ordonne pas *e* avant *d*, soit il renomme les valeurs pour éviter les anti-dépendances. Ici il y a deux anti-dépendances : *e* avec *d* et *g* avec *f* et deux dépendances de sortie : *e* avec *c* et *g* avec *e*.

L'ordonnement influe sur la durée de vie des variables, Il doit à la fois ne pas ordonner une variable avant que ses opérandes ne soient prêts pour ne pas introduire de cycles inutilisés (*idle*) et ne pas ordonner trop longtemps après que ses opérandes soient prêts pour ne pas mettre trop de pression sur les registres. Ce traitement est NP-complet, en pratique, la plupart des ordonnancements utilisés dans les compilateurs sont basés sur le *list scheduling*.

## 11.4 Ordonnancement par list scheduling

Le list scheduling est un algorithme glouton pour l'ordonnancement d'instructions dans un bloc de base. C'est plutôt une approche rassemblant un ensemble d'algorithmes. De nombreuses implementations existent qui varient essentiellement dans la priorités des instructions à choisir dans la liste.

Le list scheduling travaille sur un bloc de base. L'algorithme suit les quatre étapes suivantes :

1. Renomme les variables pour supprimer les anti-dépendances. Chaque définition reçoit un nom unique (c'est de la SSA limitée à un bloc). Cette étape n'est pas strictement nécessaire.
2. Construit le graphe de précédence  $\mathcal{P}$ . L'algorithme parcourt le bloc de la dernière instruction à la première, créant un noeud étiqueté par sa latence à chaque opération, et ajoutant un arc vers les opérations utilisant la valeur (si les anti-dépendances n'ont pas été enlevées, il faut aussi rajouter ces arcs).
3. Assigne des priorité à chaque opération. L'algorithme utilisera ces priorités comme guide pour choisir une opération à ordonnancer parmi la liste des opérations prêtes. La priorité la plus populaire est le poids du chemin critique de ce noeud à une racine.
4. Sélectionne itérativement une opération pour l'ordonnancer. La structure de donnée centrale du list scheduling est une liste d'opérations *prêtes* à être ordonnancées au cycle courant. Chaque opération dans cette liste a la propriété que ses opérandes sont disponibles. L'algorithme commence au premier cycle et sélectionne toutes les opérations qu'il peut ordonnancer dans ce cycle, ce qui initialise la liste. Il avance ensuite au prochain cycle et met à jour la liste en fonction de la nouvelle date et de la dernière opération ordonnancée.

C'est la dernière étape qui est le coeur de l'algorithme. Elle exécute une simulation abstraite de l'exécution du code, ignorant les opérations et valeurs, mais ne prenant en compte que les contraintes de temps. Voici un exemple d'algorithme qui l'implémente :

```
Cycle ← 0
Ready ← feuilles de  $\mathcal{P}$ 
Active ←  $\emptyset$ 
Tant que ( $Ready \cup Active \neq \emptyset$ )
  si  $Ready \neq \emptyset$  alors
    choisir une opération  $op$  dans  $Ready$ 
     $S(op) \leftarrow Cycle$ 
     $Active \leftarrow Active \cup op$ 
  Cycle ← Cycle + 1
  Pour chaque  $op \in Active$ 
    si  $S(op) + delay(op) \leq Cycle$  alors
      enlever  $op$  de  $Active$ 
      Pour chaque successeurs  $s$  de  $op$  dans  $\mathcal{P}$ 
        si  $s$  peut être exécuté
           $Ready \leftarrow Ready \cup s$ 
```

La qualité de l'algorithme va dépendre de l'état de la liste *Ready* à chaque étape (si elle contient une unique opération pour chaque cycle, le résultat sera optimal) et de la qualité de la priorité choisie. La complexité de l'algorithme est de  $O(n^2)$  au pire car un choix dans la liste *Ready* peut demander une passe linéaire sur toute la liste qui peut contenir toutes les opérations déjà rencontrées. On peut optimiser la mise à jour de la liste *Active* en conservant pour chaque cycles la liste des opérations qui *termine* dans ce cycle.

Pour l'initialisation, on peut prendre en compte des contraintes (*variables d'écart, slack time*) provenant des prédécesseurs du bloc. De plus, les machines modernes peuvent exécuter plusieurs opérations en parallèles sur différentes unités fonctionnelles.

Cette heuristique gloutonne pour résoudre un problème NP-complet se comporte relativement bien en général, mais elle est rarement robuste (de petits changement sur l'entrée peuvent entrainer de grosses différences dans la solution). On peut réduire ce problème en résolvant de

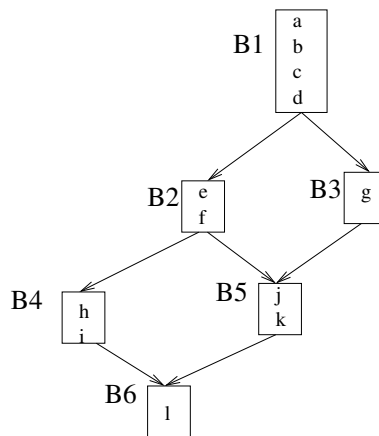
manière précise les égalités (lorsque plusieurs opérations ont la même priorité). Un bon algorithme de list scheduling aura plusieurs sortes de priorité pour résoudre les égalités. Il n'y a pas de priorité universelle, le choix dépend énormément du contexte et est difficile à prévoir.

- le rang d'un noeud peut être le poids total du plus long chemin qui le contient. Cela favorise la résolution du chemin critique d'abord et tend vers une traversée en profondeur de  $\mathcal{P}$ .
- le rang d'un noeud peut être le nombre de successeurs immédiats qu'il a dans  $\mathcal{P}$ . Cela favorise un parcours en largeur d'abord de  $\mathcal{P}$  et tend à garder plus d'opérations dans la liste *Ready*.
- le rang d'un noeud peut être le nombre total de descendant qu'il a dans  $\mathcal{P}$ . Cela amplifie le phénomène de l'approche précédente : un noeud qui produit un résultat beaucoup utilisé sera ordonnancé plus tôt.
- le rang peut être plus élevé pour les noeud ayant une longue latence. Les opérations courtes peuvent alors être utilisées pour cacher cette latence.
- Le rang peut être plus élevé pour un noeud qui contient la dernière utilisation d'une valeur. Cela diminue la pression sur les registres.

On peut aussi choisir d'exécuter l'algorithme ci-dessus en partant des racines, plutôt que des feuilles. On parle alors de *list scheduling remontant* (*backward list scheduling*), par opposition au *list scheduling descendant* (*forward list scheduling*). Une approche assez courante est d'essayer plusieurs versions du list scheduling et de garder le meilleur résultat. Si la partie critique à ordonnancer est plutôt près des feuilles, le scheduling descendant sera sans doute meilleur et réciproquement si la partie critique est près des racines.

## 11.5 Ordonnancement par région

On peut étendre le list scheduling aux blocs de base étendus, par exemple, considérons le CFG suivant :



Il contient deux blocs de base étendus non réduits à un bloc :  $(B_1, B_2, B_4)$ ,  $(B_1, B_3)$ . Ces deux EBB interfèrent par leur premier bloc :  $B_1$ . L'ordonnanceur doit éviter deux types de conflits lorsqu'il ordonnance un des EBB : Si l'ordonnanceur déplace une opération de  $B_1$  en dehors du bloc (par exemple,  $c$  après  $d$  dans le bloc  $B_2$ ). Dans ce cas,  $c$  ne sera plus sur le chemin de  $B_1$  à  $B_3$ , il faut donc introduire une copie de  $c$  dans  $B_3$  (ajout de *code de compensation*). À l'inverse, l'ordonnanceur peut remonter une opération de  $B_2$  dans  $B_1$ , Pour être sûr que cela ne modifie pas le sens du programme, il faudrait être en SSA, on peut éventuellement faire cela uniquement sur une région du graphe.

Ici l'algorithme ordonnancerait le bloc étendu  $(B_1, B_2, B_3)$  puisque c'est le plus long, puis  $B_3$  (comme partie du bloc étendu  $(B_1, B_3)$  à laquelle on a retiré  $B_1$ ), puis  $B_4, B_5$  et  $B_6$ .

La technique du *trace scheduling* utilise des informations sur le comportement du programme à l'exécution pour sélectionner les régions à ordonnancer. Il construit une trace, ou un chemin à travers le CFG qui représente le chemin le plus fréquemment exécuté. Étant donnée une trace, l'ordonnanceur applique le list scheduling à la trace entière comme pour un bloc de base étendu. Il peut être éventuellement obligé d'insérer du code de compensation aux points où le flot de

contrôle entre dans la trace. L'ordonnement des blocs de base étendus est en fait un cas dégénéré du trace scheduling.

## 11.6 Ordonnement de boucles

L'ordonnement du corps d'une boucle à un effet crucial sur les performances globales de l'exécution des programmes. Les boucles qui posent le plus de problèmes sont celles qui contiennent peu d'instructions et pour lesquelles il est difficile de cacher la latence du saut de fin de boucle. La clé pour cela est de prendre en compte simultanément plusieurs itérations successives et de faire se recouvrir dans le temps l'exécution de ces itérations successives. Par exemple si on regarde trois itérations successives, on tente donc d'ordonner les instructions de fin de l'itération  $i - 1$ , celles du milieu de l'itération  $i$  et celle du début de l'itération  $i + 1$ . Cette technique est appelé *pipeline logiciel* (*software pipelining*).

Pour atteindre ce régime permanent, il faut exécuter un prologue qui remplit le pipeline, et un épilogue à la fin de la boucle, cela peut facilement doubler la taille du code de la boucle. Considérons la boucle suivante :

```
for (i = 1; i < 200; i++)
    z[i] = x[i] * y[i]
```

Le code qui pourrait être généré est représenté ci-dessous. Le compilateur a réalisé la strength reduction pour calculer les adresses de  $x$ ,  $y$  et  $z$ , il a aussi utilisé l'adresse de  $x$  pour effectuer le test de fin de boucle (*linear function test replacement*), le dernier élément de  $x$  accédé est le 199<sup>ième</sup>, avec un décalage de  $(199 - 1) \times 4 = 792$ .

-4		<i>loadI</i>	<i>@x</i>	$\Rightarrow$	$r_{@x}$		
-3		<i>loadI</i>	<i>@y</i>	$\Rightarrow$	$r_{@y}$	//	adresses et valeurs
-2		<i>loadI</i>	<i>@z</i>	$\Rightarrow$	$r_{@z}$	//	préchargées
-1		<i>addI</i>	$r_{@x}, 792$	$\Rightarrow$	$r_{ub}$		
1	$L_1 :$	<i>loadAO</i>	$r_{sp}, r_{@x}$	$\Rightarrow$	$r_x$	//	valeurs de $x[i]$ et $y[i]$
2		<i>loadAO</i>	$r_{sp}, r_{@y}$	$\Rightarrow$	$r_y$		
3		<i>addI</i>	$r_{@x}, 4$	$\Rightarrow$	$r_{@x}$	//	incrément sur les adresses
4		<i>addI</i>	$r_{@y}, 4$	$\Rightarrow$	$r_{@y}$	//	cachant la latence des chargements
5		<i>mult</i>	$r_x, r_y$	$\Rightarrow$	$r_z$	//	le vrai travail
6		<i>cmp_LT</i>	$r_{@x}, r_{ub}$	$\Rightarrow$	$r_{cc}$	//	test pendant la latence de la multiplication
7		<i>storeAO</i>	$r_z$	$\Rightarrow$	$r_{sp}, r_{@z}$		
8		<i>addI</i>	$r_{@z}, 4$	$\Rightarrow$	$r_{@z}$		
9		<i>cbr</i>	$r_{cc}$	$\rightarrow$	$L_1, L_2$		

En supposant maintenant qu'on ait une machine avec deux unités fonctionnelles (on suppose que les *load* et *store* doivent être exécutés sur la première, en comptant toujours 3 cycle pour un *load/store*, deux cycles pour une multiplication), on peut effectuer un certain nombre de tâches en parallèle :

-2		<i>loadI</i>	<i>@x</i>	$\Rightarrow$	$r_{@x}$		<i>loadI</i>	<i>@y</i>	$\Rightarrow$	$r_{@y}$
-1		<i>loadI</i>	<i>@z</i>	$\Rightarrow$	$r_{@z}$		<i>addI</i>	$r_{@x}, 792$	$\Rightarrow$	$r_{ub}$
1	$L_1 :$	<i>loadAO</i>	$r_{sp}, r_{@x}$	$\Rightarrow$	$r_x$		<i>addI</i>	$r_{@x}, 4$	$\Rightarrow$	$r_{@x}$
2		<i>loadAO</i>	$r_{sp}, r_{@y}$	$\Rightarrow$	$r_y$		<i>addI</i>	$r_{@y}, 4$	$\Rightarrow$	$r_{@y}$
3		<i>nop</i>					<i>cmp_LT</i>	$r_{@x}, r_{ub}$	$\Rightarrow$	$r_{cc}$
4		<i>mult</i>	$r_x, r_y$	$\Rightarrow$	$r_z$		<i>addI</i>	$r_{@z}, 4$	$\Rightarrow$	$r_{@z}$
5		<i>nop</i>					<i>nop</i>			
6		<i>storeAO</i>	$r_z$	$\Rightarrow$	$r_{sp}, r_{@z}$		<i>nop</i>			
7		<i>cbr</i>	$r_{cc}$	$\rightarrow$	$L_1, L_2$		<i>nop</i>			

En fonction de l'architecture cible, le stockage de  $z$  peut ou pas bloquer la machine pendant un cycle, le corps de boucle durera donc 8 ou 9 cycles. Le corps de la boucle passe un temps significatif à attendre que les chargements arrivent. Ci dessous est représenté le code après application du pipeline logiciel.

-4	<i>loadI</i>	@x	⇒	$r_{@x}$	<i>loadI</i>	@y	⇒	$r_{@y}$	// prologue	
-3	<i>loadAO</i>	$r_{sp}, r_{@x}$	⇒	$r_{x'}$	<i>addI</i>	$r_{@x}, 4$	⇒	$r_{@x}$	// première itération (instruction 1)	
-2	<i>loadAO</i>	$r_{sp}, r_{@y}$	⇒	$r_y$	<i>addI</i>	$r_{@y}, 4$	⇒	$r_{@y}$	// première itération (instruction 2)	
-1	<i>loadI</i>	@z	⇒	$r_{@z}$	<i>addI</i>	$r_{@x}, 788$	⇒	$r_{ub}$		
0	<i>nop</i>				<i>nop</i>					
1	$L_1 :$	<i>loadAO</i>	$r_{sp}, r_{@x}$	⇒	$r_x$	<i>addI</i>	$r_{@x}, 4$	⇒	$r_{@x}$	// valeur $x[i + 1]$ et MAJ @x = $i + 2$
2		<i>loadAO</i>	$r_{sp}, r_{@y}$	⇒	$r_y$	<i>mult</i>	$r_{x'}, r_y$	⇒	$r_z$	// valeur $y[i + 1]$ , calcul $x[i] * y[i]$
3		<i>cmp_LT</i>	$r_{@x}, r_{ub}$	⇒	$r_{cc}$	<i>addI</i>	$r_{@y}, 4$	⇒	$r_{@y}$	MAJ @y = $i + 2$ , copie insérée
4		<i>storeA0</i>	$r_z$	⇒	$r_{sp}, r_{@z}$	<i>i2i</i>	$r_x$	⇒	$r_{x'}$	// stocke $z[i]$ , test iter. $i + 1$
5		<i>cbr</i>	$r_{cc}$	→	$L_1, L_2$	<i>addI</i>	$r_{@z}, 4$	⇒	$r_{@z}$	branchement itér. $i$ , MAJ @z = $i + 1$
+1		<i>mult</i>	$r_{x'}, r_y$	⇒	$r_z$	<i>nop</i>				// épilogue :fin dernière
+2		<i>storeA0</i>	$r_z$	⇒	$r_{sp}, r_{@z}$	<i>nop</i>				// itération

L'ordonnanceur a plié la boucle deux fois, le corps de boucle travaille donc sur deux itérations successives en parallèle. Le prologue exécute la première moitié de la première itération, puis chaque itération exécute la seconde moitié de l'itération  $i$  et la première moitié de l'itération  $i + 1$ . La représentation graphique permet de mieux comprendre :

<i>cycle</i>	iteration i	iteration i+1	iteration i+2
1		$r_x \leftarrow [ @x ]$	@x ← $i + 2$
2		$r_z \leftarrow r_{x'} * r_y$	$r_y \leftarrow [ @y ]$
3	<i>cmp_LT</i> $r_{ub}$		@y ← $i + 2$
4	[@z] ← $r_z$		$r_{x'} \leftarrow r_x$
5	<i>cbr</i> $L_1, L_2$	@z ← $i + 1$	
6			$r_x \leftarrow [ @x ]$
7			@x ← $i + 3$
8			$r_z \leftarrow r_{x'} * r_y$
9			$r_y \leftarrow [ @y ]$
10			@y ← $i + 3$
11			<i>cmp_LT</i> $r_{ub}$
12			[@z] ← $r_z$
13			$r'_x \leftarrow r_x$
14			<i>cbr</i> $L_1, L_2$
15			@z ← $i + 2$
16			@z ← $i + 1$
17			$r_z \leftarrow r_{x'} * r_y$
18			<i>cmp_LT</i> $r_{ub}$
19			[@z] ← $r_z$
20			<i>cbr</i> $L_1, L_2$
21			@z ← $i + 3$

On voit que les opérations de deux itérations peuvent se recouvrir facilement, ci dessous, en gras les opérations de l'étape  $i + 1$  :

6	<b><math>r_x \leftarrow [ @x ]</math></b>	<b>@x ← <math>i + 3</math></b>
7	<b><math>r_z \leftarrow r_{x'} * r_y</math></b>	$r_y \leftarrow [ @y ]$
8	<i>cmp_LT</i> $r_{ub}$	<b>@y ← <math>i + 3</math></b>
9	$r_z \leftarrow [ @z ]$	<b><math>r'_x \leftarrow r_x</math></b>
10	<i>cbr</i> $L_1, L_2$	@z ← $i + 1$

L'implémentation du pipeline logiciel a fait l'objet de nombreuses recherches et continu à être un domaine très actif.

## 12 Introduction à l'optimisation : élimination de redondances

De manière générale, le rôle de l'optimisation est de découvrir à la compilation, des informations sur le comportement à l'exécution et d'utiliser cette information pour améliorer le code généré par le compilateur. Historiquement, les premiers compilateurs comprenaient très peu d'optimisation, c'est pourquoi le terme d'*optimizing compiler* a été introduit (par opposition aux *debugging compilers* utilisés pour la mise au point puisque le code produit était proche du code source). Avec l'arrivée des architectures RISC et des mécanismes au comportement complexe (delay slot, pipeline, unités fonctionnelles multiples, opérations mémoire non bloquantes, ...), l'optimisation devint de plus en plus nécessaire et occupe maintenant une place importante dans le processus de compilation.

Il serait très long d'étudier toutes les optimisations en détail, l'ouvrage qui fait référence pour les optimisations est le livre de Muchnick [Muc]. La figure 1 donne une vision possible de l'ensemble des optimisations utilisées dans un compilateur. Bien sûr il existe d'autres manières de les agencer puisque beaucoup d'optimisations sont inter-dépendantes, l'ordre dans lequel on va les réaliser va influencer le résultat.

Sur cette figure, Les lettres indiquent les choses suivantes :

- **A** : ces optimisations sont appliquées sur le code source ou sur une représentation intermédiaire haut niveau.
- **B et C** : ces optimisations sont faites sur une représentation intermédiaire de moyen ou bas niveau.
- **D** : ces optimisations sont faites sur une représentation bas niveau qui sera assez dépendante de la machine (LLIR)
- **E** : ces optimisations peuvent être réalisées à l'édition de lien pour manipuler les objets relocables.

Les dépendances en pointillés indiquent que ces optimisations peuvent être appliquées à chaque niveau.

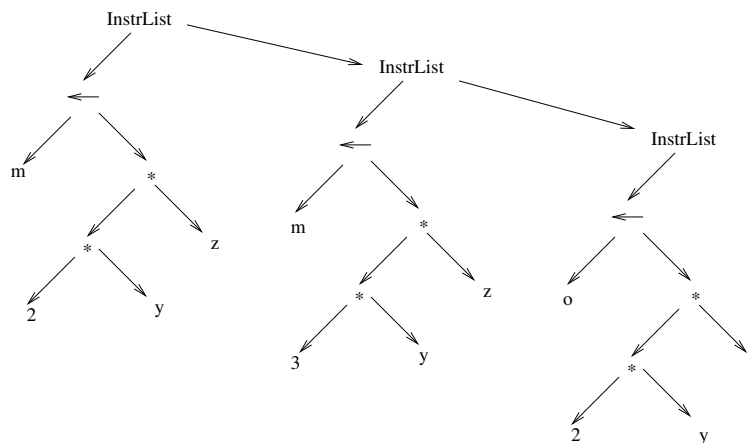
Plutôt que d'étudier beaucoup d'optimisations superficiellement, on va étudier une optimisation particulière : l'élimination d'expressions redondantes.

Considérons l'exemple suivant : on aimerait bien que le code de gauche soit remplacé par le code de droite dans lequel l'expression  $2 \times z$  n'est calculée qu'une fois.

$$\begin{array}{ll}
 m \leftarrow 2 \times y \times z & t_0 \leftarrow 2 \times y \\
 n \leftarrow 3 \times y \times z & m \leftarrow t_0 \times z \\
 o \leftarrow 2 \times y - z & n \leftarrow 3 \times y \times z \\
 & o \leftarrow t_0 - z
 \end{array}$$

### 12.1 Élimination d'expressions redondantes avec un AST

Si l'on a une représentation à base d'arbre, on va par exemple représenter le code de gauche par l'arbre suivant :



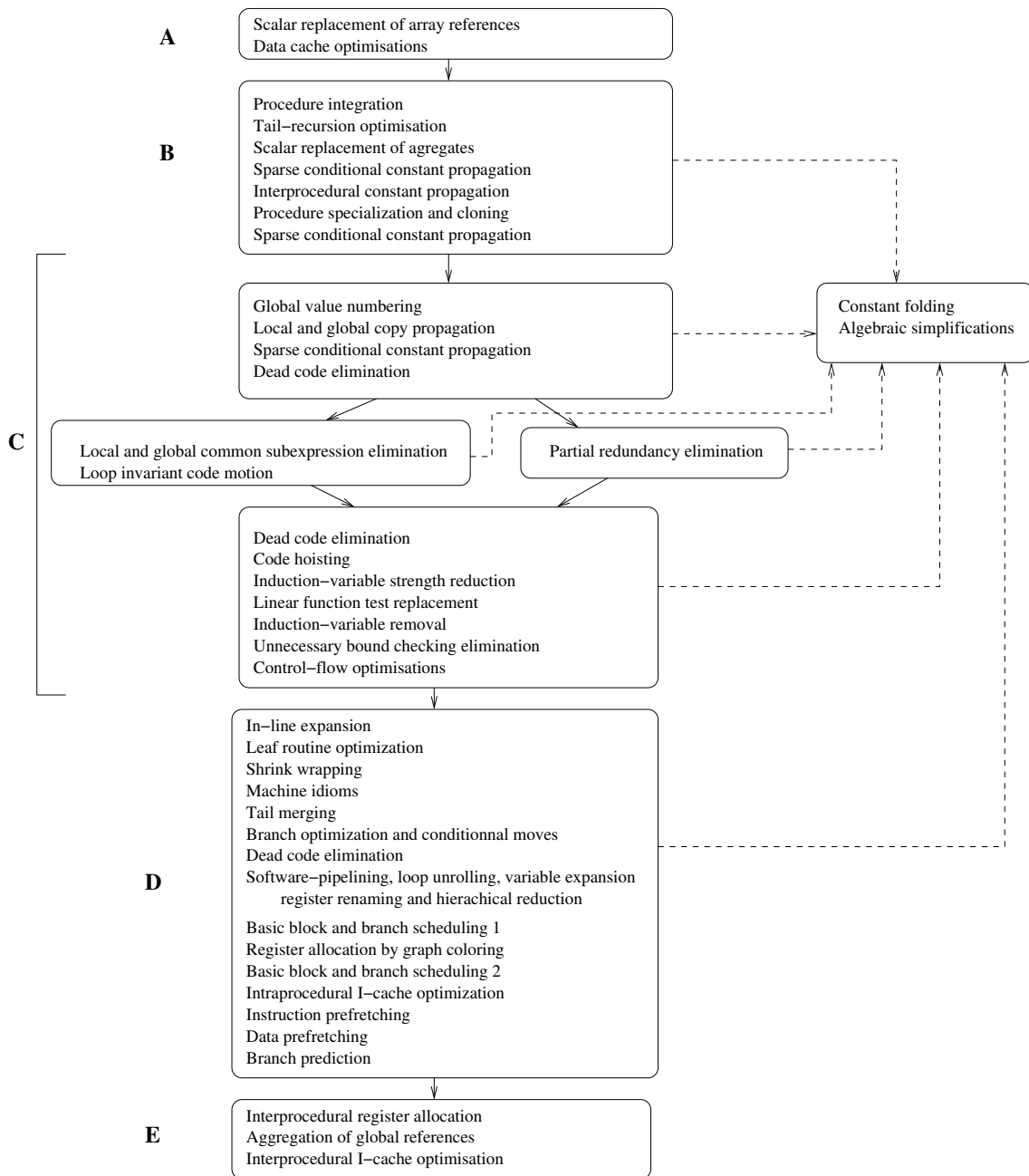
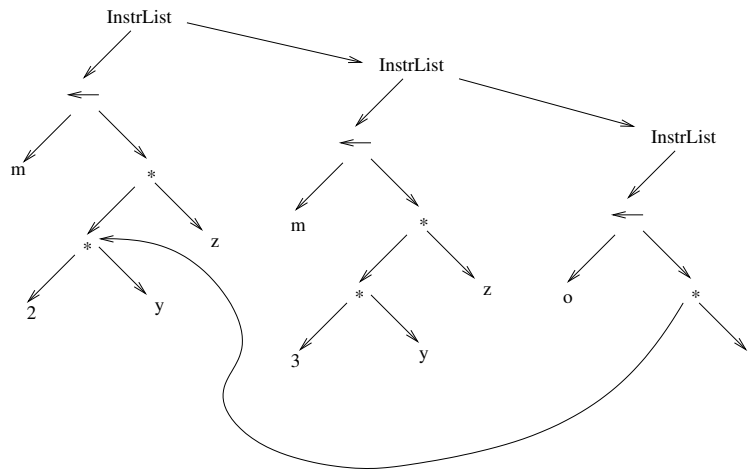


FIG. 1 – Ordre des optimisations d’après S. Muchnick



On peut alors transformer cet arbre en un graphe acyclique (directed acyclic graph : DAG), dans un tel DAG, chaque noeud ayant plusieurs parents *doit* représenter une expression redondante. Par exemple :



Notons à l'occasion que cette représentation ne permet pas de détecter la réutilisation de  $y \times z$ . pour faire cela, il faudrait faire intervenir les propriétés d'associativité et de commutativité des opérateurs.

La clé est de retrouver de tels schémas pour rendre explicite l'utilisation d'expressions redondantes. Le moyen le plus simple est de le faire lors de la construction de l'AST. Le parseur peut construire une table de hachage pour détecter les expressions identiques. Cependant un mécanisme de reconnaissance purement syntaxique ne peut suffire puisque deux expressions identiques syntaxiquement peuvent représenter deux valeurs différentes, il faut un mécanisme pour refléter l'influence des assignations.

Le mécanisme utilisé est simple : à chaque variable on associe un compteur qui est incrémenté lors des assignation sur la variable. Dans le constructeur de la table de hachage, on rajoute le compteur à la fin du nom de la variable, cela permet de ne pas identifier des expressions dont les valeurs des sous-expressions ont changé.

Le problème vient une fois encore des pointeurs. un assignation telle que  $*p = 0$  peut potentiellement modifier n'importe quelle variable. Si le compilateur n'arrive pas à connaître les endroits où peut pointer  $p$ , il est obligé d'incrémenter les compteurs de toutes les variables. Une des motivations de l'analyse de pointeurs est de réduire l'ensemble des variables potentiellement modifiées par une assignation de pointeur.

Cette approche fonctionne sur les représentations intermédiaires à base d'arbres, elle permet aussi de diminuer la taille de la représentation intermédiaire.

## 12.2 Valeurs numérotées

Pour les représentations intermédiaires linéaires, la technique utilisée est celle des valeurs numérotées (*value numbering*). Cette méthode assigne un nombre unique à chaque valeur calculée à l'exécution avec la propriété que deux expressions  $e_i$  et  $e_j$  ont le même *numéro de valeur* si et seulement si elles s'évalueront toujours à la même valeur dans toutes les exécutions possibles.

Le principe est similaire à la technique précédente : le compilateur utilise une table de hachage pour identifier les expressions qui produisent la même valeur. La notion de "redondant" diffère un peu : deux opérations ont le même numéro de valeur si elles ont les mêmes opérateurs et leurs opérands ont les mêmes numéros de valeur. L'assignation de pointeur a toujours l'effet désastreux d'invalider tous les numéros de valeurs des variables pouvant être pointées par le pointeur.

L'algorithme est assez simple quand on se limite à un bloc de base : à chaque opération, on construit une clé de hachage pour l'expression à partir des numéros de valeurs des opérands. Si cette clé existe, on peut réutiliser une valeur sinon on introduit la valeur dans la table de hachage.

On le fait tourner sur l'exemple suivant :

$a \leftarrow b + c$	$a^3 \leftarrow b^1 + c^2$	$a \leftarrow b + c$
$b \leftarrow a - d$	$b^5 \leftarrow a^3 - d^4$	$b \leftarrow a - d$
$c \leftarrow b + c$	$c^6 \leftarrow b^5 + c^2$	$c \leftarrow b + c$
$d \leftarrow a - d$	$d^5 \leftarrow a^3 - d^4$	$d \leftarrow b$
Code original	numéro de valeurs	Code après réécriture

Cet algorithme de base peut être augmenté de plusieurs manières. Par exemple, on peut prendre en compte la commutativité des opérateurs en ordonnant les opérandes des opérateurs commutatifs d'une manière systématique, on peut aussi remplacer les expressions constantes par leurs valeurs, découvrir les identités algébriques etc. Un algorithme possible pour une représentation de type Iloc serait :

Pour chaque opération  $i$  dans le bloc, l'opération  $i$  étant de la forme  $op_i x_1, x_2, \dots, x_{k-1} \Rightarrow x_k$

- 1) récupérer les numéros de valeurs de  $x_1, x_2, \dots, x_{k-1}$
- 2) si toutes les entrées de  $op_i$  sont constantes  
évaluer la constante et enregistrer sa valeur
- 3) Si l'opération est une identité algébrique  
la remplacer par une opération de copie
- 4) si  $op_i$  est commutative  
trier les opérandes par numéro de valeur
- 5) Construire la clé de hachage d'après l'opérateur et les numéros de valeur de  $x_1, x_2, \dots, x_{k-1}$
- 6) Si la valeur existe déjà dans la table  
remplacer l'opération  $i$  par une copie

Sinon

assigner un nouveau numéro de valeur à la clé et l'enregistrer pour  $x_k$ .

Dans cette approche le nom des variables a une importance primordiale. Considérons l'exemple suivant :

$a \leftarrow x + y$	$a^3 \leftarrow x^1 + y^2$	$a \leftarrow x + y$
$b \leftarrow x + y$	$b^3 \leftarrow x^1 + y^2$	$b \leftarrow a$
$a \leftarrow 17$	$a^4 \leftarrow 17^4$	$a \leftarrow 17$
$c \leftarrow x + y$	$c^3 \leftarrow x^1 + y^2$	$c \leftarrow x + y$
Code original	numéro de valeurs	Code après réécriture

La valeur  $x+y$  n'existe plus lorsque l'on arrive à la définition de  $c$  car elle avait été enregistrée pour  $a$  qui a été écrasée entre temps. La solution à ce problème passe par le forme SSA. La réécriture du code en forme SSA donnera :

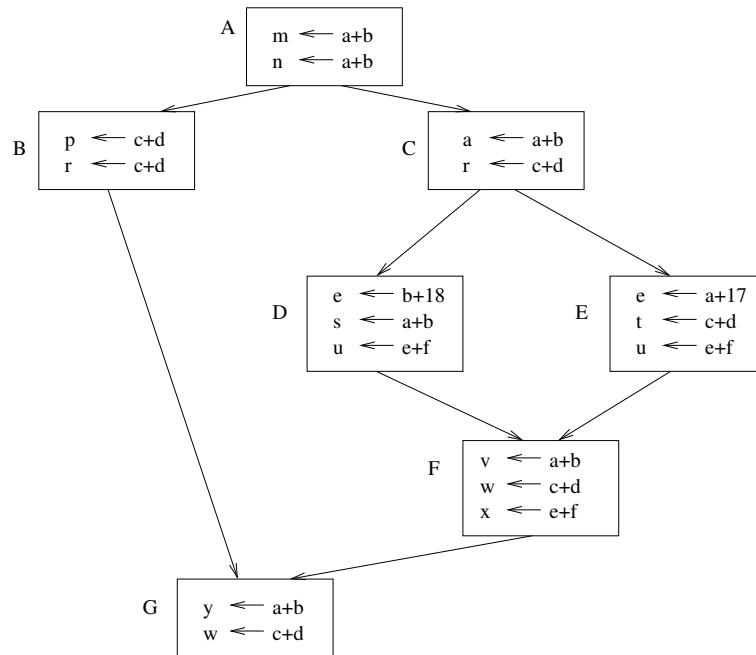
$$\begin{aligned}
 a_0 &\leftarrow x_0 + y_0 \\
 b_0 &\leftarrow x_0 + y_0 \\
 a_1 &\leftarrow 17 \\
 c_0 &\leftarrow x_0 + y_0 \\
 \text{Code en SSA}
 \end{aligned}$$

On pourra alors non seulement reconnaître que les trois utilisations de  $x + y$  sont les mêmes mais aussi facilement supprimer les variables  $b_0$  et  $c_0$  dans le reste du code car on sait qu'elles ne risquent pas de changer de valeur.

### 12.3 Au delà d'un bloc de base

Dès que l'on sort d'un bloc de base, les choses deviennent nettement plus compliquées. On rappelle la définition d'un bloc de base (dans un IR linéaire) : c'est une suite d'instructions qui ne comporte aucun saut (sauf la dernière instruction) et aucun autre point d'entrée que la première instruction (un bloc de base a donc la propriété que, sauf si une exception est levée, toutes les opérations du bloc s'exécutent à chaque exécution du bloc).

Considérons l'exemple suivant :

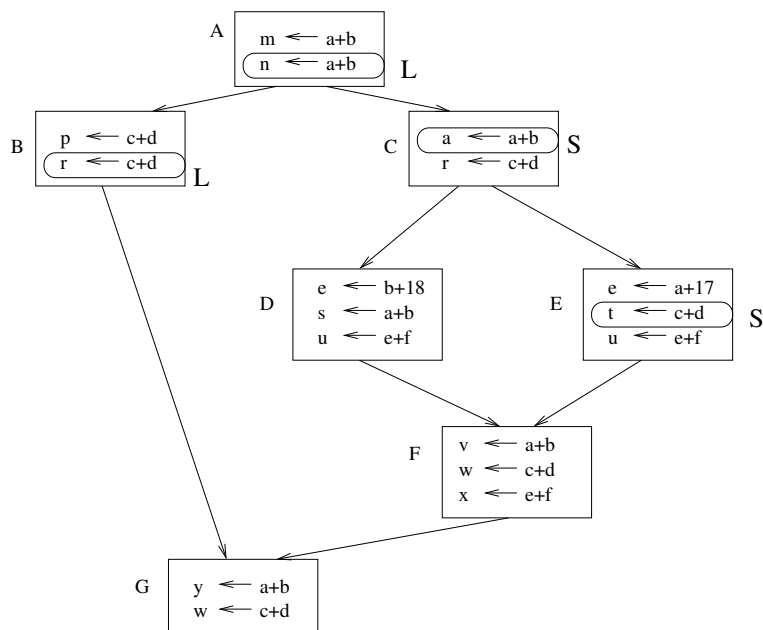


On distingue 4 types de méthodes :

1. Les méthodes locales travaillent sur les blocs de base.
2. Les méthodes superlocales opèrent sur des *blocs de bases étendus*. Un bloc de base étendu  $B$  est un ensemble de blocs de base  $\beta_1, \beta_2, \dots, \beta_n$  où  $\beta_1$  a de multiples prédécesseurs et chaque  $\beta_i, 2 \leq i \leq n$  a  $\beta_{i-1}$  comme unique prédécesseur. Un bloc de base étendu a donc un unique point d'entrée mais plusieurs points de sorties. Sur l'exemple ci-dessus,  $\{A, B\}$ ,  $\{A, C, D\}$ ,  $\{A, C, E\}$ ,  $\{F\}$ ,  $\{G\}$  sont des blocs de base étendus (*extended basic blocs, EBB*).
3. Les méthodes régionales travaillent sur des régions de blocs mais pas sur une procédure entière, par exemple l'ensemble des blocs de base formant une boucle. Elle prennent en compte les *point de synchronisation* ou confluent deux flots de contrôle entrant (ici en  $F$  ou  $G$  par exemple).
4. Les méthodes globales ou *intra-procédurales* travaillent sur l'ensemble de la procédure et utilisent l'analyse flot de donnée pour découvrir les informations transmises entre blocs
5. les méthodes inter-procédurales.

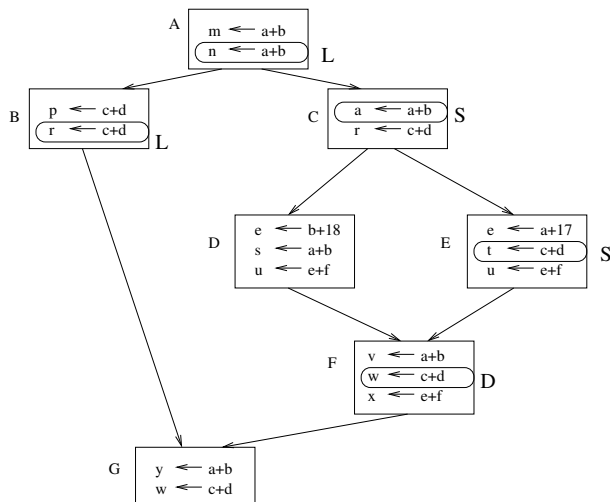
**Approche super-locale** On peut par exemple calculer la table de hachage correspondant au bloc  $A$ , puis l'utiliser comme état initial pour numéroter les valeurs du bloc  $C$ , puis utiliser la table résultante comme entrée pour le bloc  $D$ . Il faudra ensuite recalculer la table pour le bloc  $A$  afin de l'utiliser pour le calcul sur le super-bloc  $\{A, B\}$ .

On pourra éviter les calculs redondants de table en utilisant une table de hachage imbriquée lexicalement comme pour les accès aux variables de différentes portées. L'exemple courant est représenté ci-dessous. L'ordre de recherche des expressions redondantes pourra être :  $A, B, C, D, E, F, G$ . Les assignations entourées représentent les calculs redondants découverts qui peuvent être supprimées (lettre L pour méthode locale, S pour méthode super-locale).



La méthode super-locale améliore la détection d'expressions redondantes, mais elle est limitée aux super-blocs : l'expression  $a + b$  dans le bloc  $F$  n'est pas détectée comme étant redondante. Un autre cas qui n'est pas détecté est le calcul de  $e + f$  dans le bloc  $F$ . Le calcul est redondant avec soit  $e + f$  du bloc  $D$  soit  $e + f$  du bloc  $E$  suivant le chemin pris par le flot de contrôle, mais ces deux valeurs ne sont pas égales.

**Approche régionale** En arrivant au bloc  $F$ , on ne peut utiliser aucune des tables de  $D$  ou  $E$  car on ne sait pas par où est passé le contrôle. En revanche, on peut utiliser la table résultant du passage de l'algorithme sur le bloc étendu  $\{A, C\}$ . En effet, tout programme passant par le bloc  $F$  est obligatoirement passé par les blocs  $A$  et  $C$ . On peut donc utiliser la table de hachage du bloc  $C$  comme initialisation pour le traitement du bloc  $F$ . Attention cependant, il faut vérifier que les valeurs présentes dans la table de hachage du bloc  $C$  ne sont pas *tuées* par un des blocs entre  $C$  et  $F$ . Par exemple ici, comme  $a$  est redéfini dans le bloc  $C$ , l'expression  $a + b$  du bloc  $A$  est tuée et ne peut être réutilisée dans le bloc  $G$ . Pour repérer cela, on peut soit passer en SSA soit maintenir une liste des variables modifiées par bloc et mettre à jour la table de hachage en conséquence. Ce principe est représenté ci-dessus (le  $D$  indique les instructions retirées par des considérations de domination) :



Le bloc utilisé pour initialiser une table de hachage est le *dominateur immédiat* de ce bloc. On dit qu'un bloc  $B_1$  domine un bloc  $B_2$  si, dans le graphe de contrôle de flot, tous les chemins allant

de l'entrée du programme au bloc  $B_2$  passent par  $B_1$ . On note  $B_1 \succeq B_2$ , si  $B_1 \neq B_2$ ,  $B_1$  domine strictement  $B_2$  ( $B_1 \succ B_2$ ). C'est une relation d'ordre partielle, l'ensemble des dominateurs d'un bloc  $B_1$  est noté  $Dom(B_1)$ . Par exemple,  $Dom(F) = \{A, C, F\}$ . Le dominateur immédiat de  $B_1$  est le dominateur strict de  $B_1$  le plus proche de  $B_1$  :  $iDom(F) = C$ .

On voit assez facilement que le dominateur immédiat est celui qui est utilisé pour l'algorithme de détection de redondance : pour démarrer l'analyse d'expressions redondantes d'un bloc  $B_1$  on utilise la table de hachage du bloc  $iDom(B_1)$ .

**Détection de redondances globale** Cette approche marche bien tant qu'il n'y a pas de circuits dans le graphe de contrôle de flot. On sent bien que le processus se mord la queue puisque pour initialiser la table de hachage d'un bloc qui se trouve sur un circuit, on a besoin des tables de blocs qui vont eux-mêmes être initialisés avec la table du bloc courant.

Pour résoudre cela on décompose l'algorithme de détection d'expressions redondantes en deux phases : une première phase qui identifie toutes les expressions redondantes puis une phase qui réécrit le code. La phase de détection se fait en utilisant une analyse de flot de donnée (*data flow analysis*) pour calculer l'ensemble d'expressions *disponibles* à l'entrée de chaque bloc. Le terme *disponible* a une définition précise :

- Une expression  $e$  est *définie* (*defined*) à un point  $p$  du graphe de contrôle de flot (CFG) si sa valeur est calculée à  $p$ .
- Une valeur est *tuée* (*killed*) à un point  $p$  du CFG si un de ses opérande est défini à  $p$ .
- une expression est *disponible* (*available*) à un point  $p$  du CFG si chaque chemin qui mène à  $p$  contient une définition de  $e$ , et  $e$  n'est pas n'est pas tué entre cette définition et  $p$ .

Une fois qu'on a l'information qu'une valeur est disponible, on peut créer une variable temporaire et insérer des copies au lieu de recalculer.

Le compilateur va donc annoter chaque bloc  $B$  avec un ensemble  $Avail(B)$  qui contiendra l'ensemble des expressions disponibles à l'entrée de  $B$ . Cet ensemble peut être spécifié par un ensemble d'équations déduites de la définition de disponibilité.

On définit, pour chaque bloc  $B$ , l'ensemble  $Def(B)$  contenant l'ensemble des expressions définies dans  $B$  et non tuées ensuite dans  $B$ . Donc  $e \in Def(B)$  si et seulement si  $B$  évalue  $e$  et aucun des opérandes de  $e$  n'est défini entre l'évaluation de  $B$  et la fin du bloc.

Ensuite, on définit l'ensemble  $NotKilled(B)$  qui contient les expressions qui ne sont pas tuées par une définition dans  $B$ . C'est à dire que si  $e$  est disponible en entrée de  $B$  ( $e \in Avail(B)$ ) et que  $e \in NotKilled(B)$  alors  $e$  est disponible à la fin de  $B$ .

Une fois que ces deux ensembles sont calculés, on peut écrire une condition que doit vérifier l'ensemble  $Avail(B)$  :

$$Avail(B) = \bigcap_{p \in pred(B)} (Def(p) \cup (Avail(p) \cap NotKilled(p))) \quad (3)$$

Le terme  $Def(p) \cup (Avail(p) \cap NotKilled(p))$  spécifie l'ensemble des expressions qui sont disponibles en sortie de  $p$ . Cette équation ne *définit* pas  $Avail(B)$  à proprement parler (sauf si le CFG ne contient pas de circuit) car  $Avail(B)$  est peut-être défini en fonction de lui-même. On sait simplement que les ensembles  $Avail(B)$  forment un *point fixe* de cette équation. On va donc les trouver par un algorithme de point fixe qui consiste à itérer cette équation sur tous les blocs de base  $B$  du CFG jusqu'à ce que les ensembles  $Avail(B)$  ne changent plus.

Le calcul des ensembles  $Def(p)$  est immédiat, le calcul des ensembles  $NotKilled(p)$  est plus délicat. Voici un algorithme qui calcule les deux en même temps :

```
//On travaille sur un bloc B avec les opérations o1, o2, ..., ok.
Killed ← ∅
Def(B) ← ∅
for i=k to 1
  //supposons que oi soit : "x ← y + z"
  si (y ∉ Killed) et (z ∉ Killed) alors
    ajouter "x + y" à Def(B)
  ajouter x à Killed
```

```

NotKilled(B) ← {toutes les expressions}
Pour chaque expression e
  Pour chaque variables v ∈ e
    si v ∈ Killed alors
      NotKilled(B) ← NotKilled(B) − e

```

Pour calculer les ensembles *Avail* on peut utiliser un simple algorithme itératif (on suppose que les blocs du programme ont été numérotés de  $b_0$  à  $b_n$ ).

```

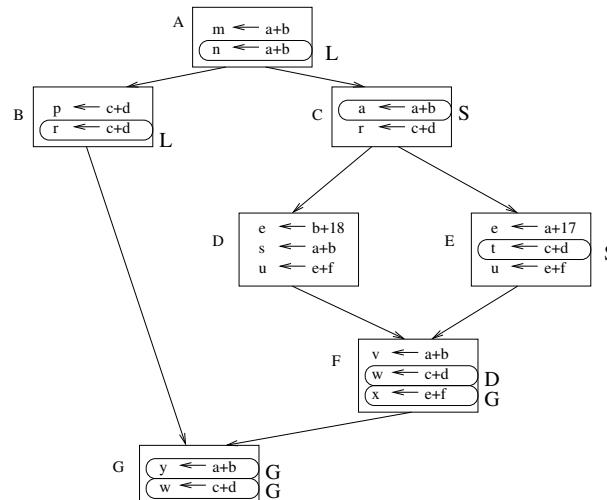
for 0 ≤ i ≤ n
  Calculer Def(bi) et NotKilled(bi)
  Avail(bi) ← ∅
Changed ← True
Tant que (Changed)
  Changed ← False
  for 0 ≤ i ≤ n
    OldValue ← Avail(bi)
    Avail(bi) := ∩p∈pred(bi) (Def(p) ∪ (Avail(p) ∩ NotKilled(p)))
    si Avail(bi) ≠ OldValue alors
      Changed ← True

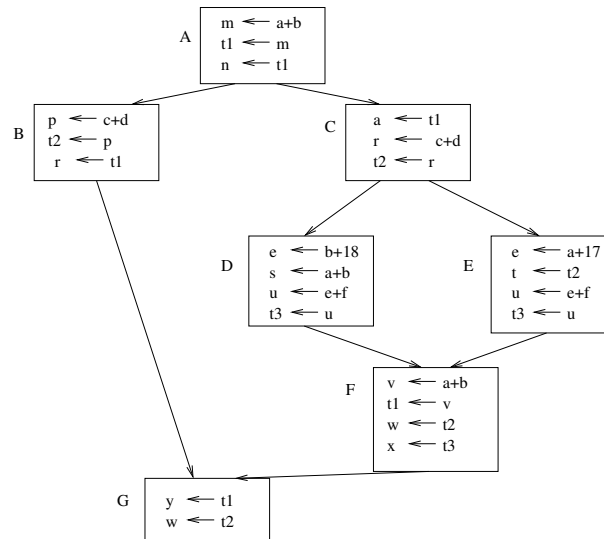
```

On peut montrer que ce problème à une structure spécifique qui en fait un calcul de point fixe. Les ensembles *Avail* ne vont faire que grossir jusqu'à ce qu'ils ne bougent plus. Ces ensembles  $Avail(b_i)$  réalisent l'équation (3) (il sont en fait obtenus en itérant de nombreuses fois cette équation dans un certain ordre). Les équations décrivant les ensembles *Avail* plus haut sont des équations d'un problème d'analyse de flot de donnée global (global data flow analysis problem). On explore ce concept plus en détail dans la section suivante.

La phase de réécriture de l'élimination de redondances doit remplacer toute expression étiquetée comme redondante par une copie de la valeur calculée antérieurement (on suppose en général que ces copies peuvent être éliminées par une passe ultérieure d'optimisation).

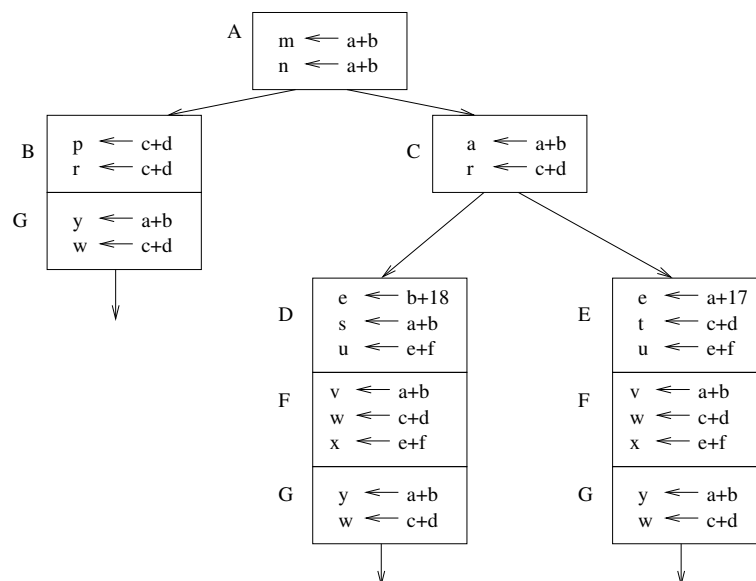
Le traitement est effectué en deux passes linéaires sur le code : la première passe identifie les endroits où des expressions redondantes sont recalculées. La table des valeurs de chaque bloc  $b$  est initialisée avec  $Avail(b)$ , chaque expression redondante  $e$  découverte est redéfinie avec une copie d'un nouveau temporaire  $temp_e$  associé à  $e$ . La deuxième passe initialisera le temporaire dans tous les blocs de base où l'expression  $e$  est définie. Le résultat final sur sur notre exemple est illustré ci-dessous à gauche (la lettre  $G$  signifiant global). A droite on a représenté le code après la passe de réécriture. On suppose en général qu'une passe d'optimisation ultérieure essayera d'enlever des copies par substitution.





**Pour aller plus loin** La méthode utilisant l'analyse data flow pour résoudre ce problème n'est pas forcément la meilleure. On peut essayer de pousser encore les méthodes super-locales qui sont relativement simples. Pour éviter les problèmes posés par les expressions  $e + f$  dans  $F$  et  $c + d$  dans  $G$ , on peut faire de la réplication de code. il y a deux méthodes utilisées : la réplication de blocs (*cloning*) et le déroulement de procédure (*inline substitution*).

La réplication de bloc permet de supprimer des points de convergence du graphe qui faisaient perdre l'information des expressions calculées. Le compilateur va simplement rajouter une copie des blocs  $F$  et  $G$  à la fin des bloc  $C$  et  $D$  ainsi qu'une copie du bloc  $G$  à la fin du bloc  $B$ .



Malgré l'expansion de code, cette transformation a plusieurs avantages : elle crée des blocs plus longs qui se comportent mieux pour diverses optimisations, elle supprime des branchements à l'exécution et elle crée des opportunités d'optimisation qui n'existaient pas avant à cause de l'ambiguïté introduite par la convergence des flots de contrôle.

le déroulement de procédure (*inline substitution*) consiste à remplacer le code d'un appel de procédure par une copie du code de la procédure appelée. Cela permet de faire une optimisation plus précise : le contexte de l'appel de procédure est transmis au corps de la procédure. En particulier, certaines branches complètes peuvent être supprimées du fait que certaines conditions utilisées dans le contrôle du flot peuvent être résolues statiquement. Il faut utiliser cette transformation avec précaution, en particulier si la pression sur les registres est forte.

## 13 Quelques transformations de code

### 13.1 Élimination de code mort

Si le compilateur peut prouver qu'une opération n'a aucun effet extérieur visible, l'opération est dite *morte* (le compilateur peut la supprimer). Cette situation arrive dans deux cas : soit l'opération est *inatteignable* (aucun chemin possible dans le flot de contrôle ne contient l'opération), soit l'opération est *inutile* (aucune opération n'utilise le résultat).

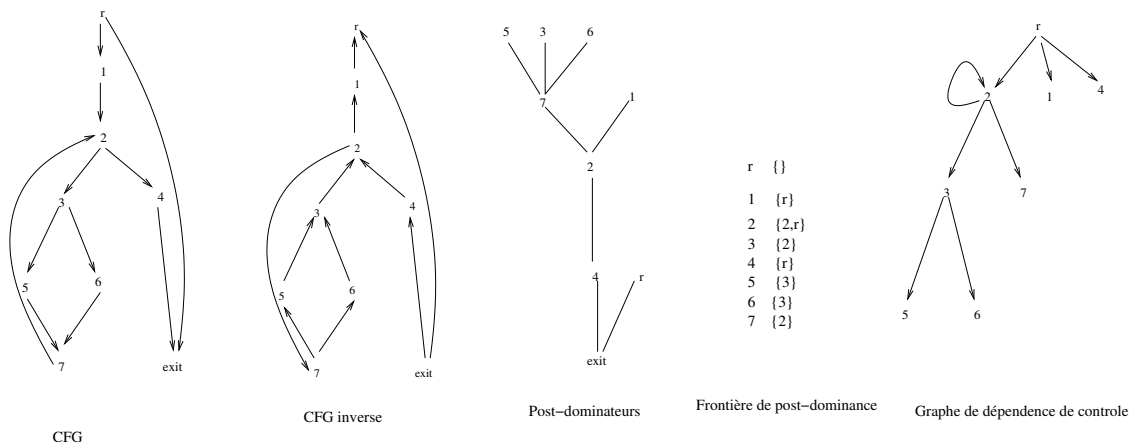
**Élimination de code inutile** L'élimination de code inutile se fait un peu comme la gestion du tas (*mark-sweep collector*), elle agit en deux passes. La première passe commence par repérer les opérations *critiques* : entrées/sorties, appels de procédure, codes d'entrée et de sortie des procédures, passages de résultats et marque toutes ces opérations comme utiles. Ensuite elle cherche les définitions des opérations définissant les opérandes de ces opérations utiles, les marque et remonte ainsi récursivement de définition en définition. À la fin de la première passe, toutes les opérations non marquées sont inutiles. En général on pratique cela sur une forme SSA car on sait qu'une utilisation correspond à une seule définition. La deuxième passe va simplement éliminer les opérations inutiles.

Lors de la première passe, les seules opérations dont le traitement est délicat sont les branchements et les sauts. Tous les sauts sont considérés comme utiles, les branchements ne sont considérés comme utiles que si l'exécution d'une opération utile dépend de leur présence. Pour relier une opération au branchement qui la rend possible, on utilise la notion de *dépendance de contrôle*. On définit les dépendances de contrôle grâce à la notion de *postdomination*.

Dans le CFG, un noeud  $j$  postdomine un noeud  $i$  si chaque chemin de  $i$  vers le noeud de sortie du CFG passe par  $j$  (i.e.  $j$  domine  $i$  dans le CFG inversé). Un noeud  $j$  est dépendant d'un noeud  $i$  du point de vue du contrôle si et seulement si :

1. Il existe un chemin non nul de  $i$  à  $j$  dans lequel  $j$  postdomine chaque noeud sur le chemin après  $i$ .
2.  $j$  ne postdomine pas strictement  $i$ . Un arc quitte  $i$  et mène à un chemin ne contenant pas  $j$  qui va vers le noeud de sortie de la procédure.

Notons que cette notion correspond à la *frontière de postdominance* qui est exactement la frontière de dominance du graphe inversé (*frontière de dominance inversée, reverse dominance frontier*). L'idée derrière les dépendances de contrôle est la suivante : si un noeud  $j$  postdomine un noeud  $i$ , alors quel que soit le résultat de l'évaluation d'un branchement dans le noeud  $i$ , le noeud  $j$  sera atteint quand même, donc les branchements du noeud  $i$  ne sont pas utiles pour atteindre le noeud  $j$ . Voici un exemple :

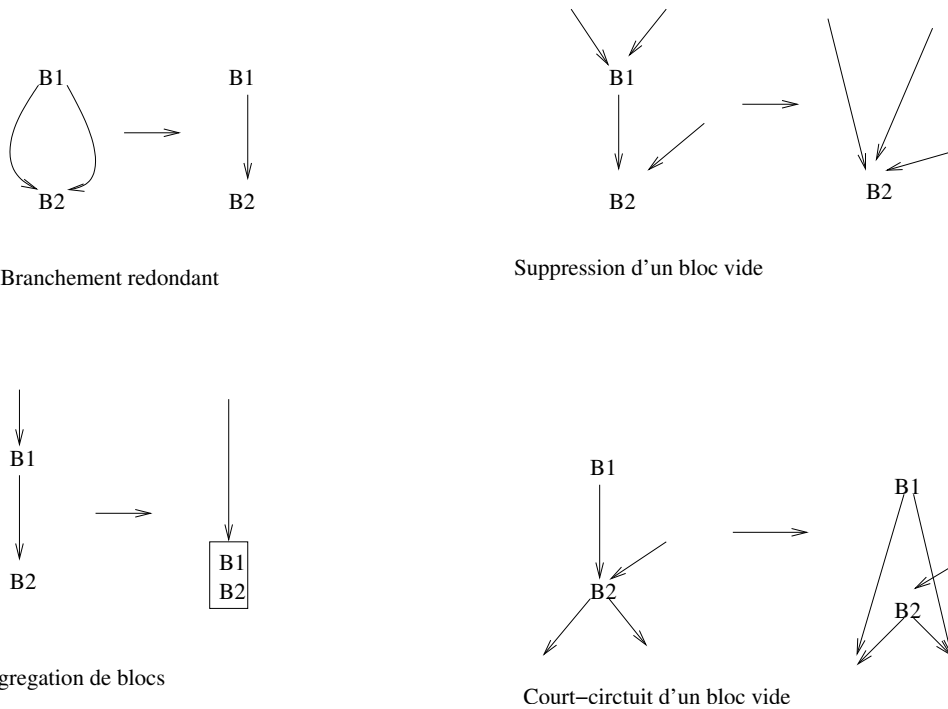


Lorsqu'une opération d'un bloc  $b_i$  est marquée comme utile par la première passe, on visite chaque bloc sur la frontière de dominance inversée de  $b_i$ . Chacun de ces blocs se finit par un branchement qui est utile pour arriver au bloc  $b_i$ , donc on marque les branchements de fin de blocs comme utiles pour tous ces blocs et on les rajoute à la liste d'instructions à traiter.

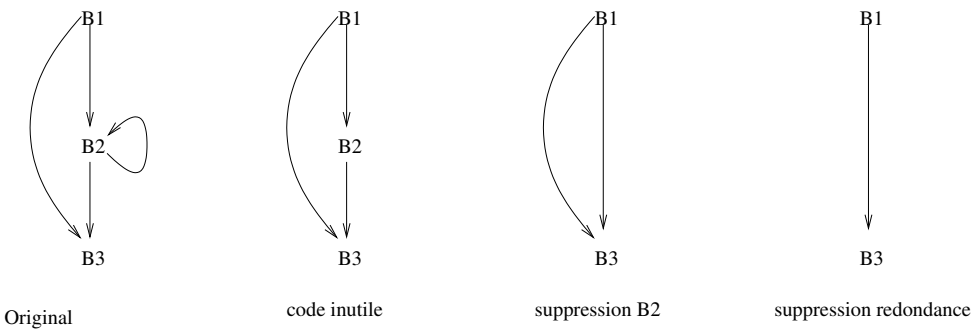


Si un branchement est non marqué alors ses successeurs jusqu'à son post-dominateur immédiat ne contiennent aucune opération marquée. Dans la deuxième phase, chaque branchement de fin de bloc non marqué est remplacé par un saut au premier post dominateur qui contient une opération marquée comme utile. Le graphe résultant peut éventuellement contenir des blocs vides qui peuvent être enlevés par la suite.

**Élimination d'opérations de contrôle de flot inutile** Cette opération fait appel à quatre transformations (voir schéma ci-dessous). Le court circuit (*branch hoisting*) arrive lorsque l'on rencontre un bloc vide en fin de bloc, on peut alors recopier le branchement dans les blocs précédents. Il faut bien distinguer les branchement des sauts, si il y a un saut en fin de bloc d'un bloc vide, on peut supprimer le bloc.



Aucune de ces transformations ne permet de supprimer une boucle vide : considérons par exemple le graphe ci-dessous et supposons que  $B2$  soit vide. Le branchement de  $B2$  n'est pas redondant,  $B2$  ne termine pas par un saut, il ne peut donc pas être combiné avec  $B3$ . son prédécesseur  $B1$  finit par un branchement, il ne peut pas être fusionné avec et il ne peut pas être court-circuité.



Une combinaison des deux éliminations vues ici peut supprimer la boucle, l'élimination de code mort ne marquera pas le branchement finissant  $B2$  comme utile ( $B3$  postdomine  $B2$  donc  $B2 \notin RDF(B3)$ ). le branchement sera alors remplacé par un saut et les opérations de simplification du contrôle de flot feront le reste.

**Élimination de code inateignable** L'élimination des blocs ne se trouvant sur aucun chemin de contrôle est facile après la première passe du processus précédent.

### 13.2 Strength reduction

La transformation désignée par *strength reduction* remplace une suite répétitive d'opérations complexes par une suite répétitive d'opérations simples qui calculent la même chose. Elle est essentiellement utilisée pour les accès aux tableaux dans les boucles.

Considérons la boucle simple suivante :

```
sum ← 0
for i ← 1 to 100
    sum ← sum + a(i)
```

Un exemple code Iloc pouvant être généré après passage en SSA est donné ici (on suppose que le tableau est déclaré entre en  $a[1..100]$ ) :

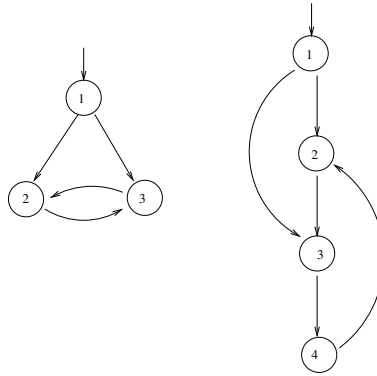
	<i>loadI</i>	0	⇒	$r_{s0}$		$r_{s0}$	←	0
	<i>loadI</i>	1	⇒	$r_{i0}$		$r_{i0}$	←	1
	<i>loadI</i>	100	⇒	$r_{100}$		$r_{100}$	←	100
$l_1$	<i>phi</i>	$r_{i0}, r_{i2}$	⇒	$r_{i1}$		$l_1$ $r_{i1}$	←	$phi(r_{i0}, r_{i2})$
	<i>phi</i>	$r_{s0}, r_{s2}$	⇒	$r_{s1}$		$r_{s1}$	←	$phi(r_{s0}, r_{s2})$
	<i>subI</i>	$r_{i1}, 1$	⇒	$r_1$		$r_1$	←	$r_{i1} - 1$
	<i>multI</i>	$r_1, 4$	⇒	$r_2$		$r_2$	←	$r_1 * 4$
	<i>addI</i>	$r_2, @a$	⇒	$r_3$		$r_3$	←	$r_2 + @a$
	<i>load</i>	$r_3$	⇒	$r_4$		$r_4$	←	$mem(r_3)$
	<i>add</i>	$r_4, r_{s1}$	⇒	$r_{s2}$		$r_{s2}$	←	$r_4 + r_{s1}$
	<i>addI</i>	$r_{i1}, 1$	⇒	$r_{i2}$		$r_{i2}$	←	$r_{i1} + 1$
	<i>cmp_LE</i>	$r_{i2}, r_{100}$	⇒	$r_5$		$r_5$	←	$r_{i2} <? r_{100}$
	<i>cbr</i>	$r_5$	→	$l_1, l_2$		$cbr$ $r_5$	→	$l_1, l_2$
$l_2$	...					$l_2$	...	

Naturellement, à chaque itération de la boucle, l'adresse de  $a(i)$  est recalculée et les registres  $r_1, r_2$  et  $r_3$  sont là uniquement pour calculer cette adresse. L'idée de la transformation de *strength reduction* est de calculer directement cette adresse à partir de sa valeur à l'itération précédente. En effet, les valeurs prises par  $r_3$  lors des itérations successives sont :  $@a, @a + 4, @a + 8, \dots, @a + 396$ . En plus du gain lié aux registres supprimés et au remplacement de la multiplication par une addition, cette transformation permet d'avoir une forme de code plus agréable pour la suite, en particulier si le jeu d'instructions contient des modes d'adressage avec auto-incrément des opérateurs. Nous présentons ici un algorithme simple pour la transformation de *strength reduction* appelé OSR.

**Graphe réductible** Pour optimiser les boucles, il faut avoir une définition précise de ce que c'est qu'une boucle. Une boucle est un ensemble de noeud  $S$  du CFG qui contient un noeud d'entrée  $h$  (*header*) avec les propriétés suivante :

- De n'importe quel noeud de  $S$  il existe un chemin dans  $S$  arrivant à  $h$ .
- Il y a un chemin de  $h$  à n'importe quel noeud de  $S$ .
- il n'y a aucun arc venant de l'extérieur de  $S$  et aboutissant à un autre noeud de  $S$  que  $h$ .

On peut montrer que c'est équivalent à dire que que  $S$  est une composante fortement connexe du CFG et que  $h$  domine tous les noeuds de  $S$ . Une boucle peut donc avoir plusieurs sorties mais une seule entrée. Voici deux exemples de graphes ne correspondant pas à des boucles.



La notion de boucle ainsi définie correspond exactement à la notion de graphe réductible. Un graphe est réductible si il peut être ramené à un seul sommet par application répétitive de la transformation suivante : un sommet qui a un seul prédécesseur peut être fusionné avec ce prédécesseur. Les graphes présentés ci dessus ne sont pas réductibles. On peut montrer que les graphes de contrôle de flot obtenus à partir de programmes contenant des mécanismes de contrôle structurés (c'est à dire tout -même les break- sauf les goto) sont des graphes réductibles. On suppose ici que l'on travaille sur des graphes réductibles.

**Algorithm OSR** L'algorithme va rechercher des points où une certaine opération (par exemple une multiplication) s'exécute à l'intérieur d'une boucle et dont les arguments sont :

1. une valeur qui ne varie pas lors des différentes itérations de la boucle (*constante de région, region constant*), et
2. une valeur qui varie systématiquement d'itération en itération (*variable d'induction*).

Une telle opération est appelée une *opération candidate*. Pour simplifier l'exposé de l'algorithme, on ne va considérer que les opérations candidates ayant la forme suivante :

$$x \leftarrow i \times j \quad x \leftarrow j \times i \quad x \leftarrow i \pm j \quad x \leftarrow j + i$$

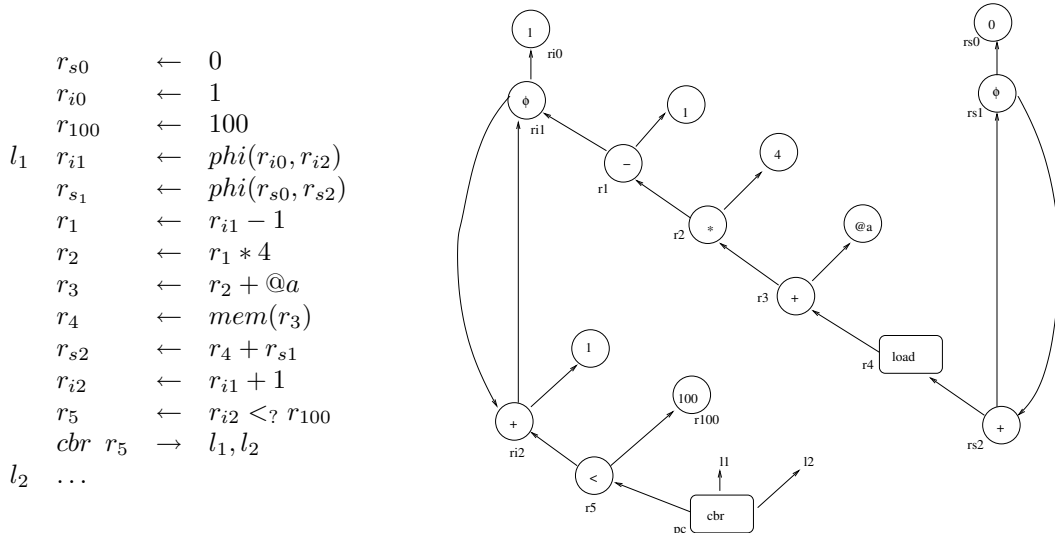
où  $i$  est une variable d'induction et  $j$  est une constante de région. Si le code est en SSA, le compilateur peut déterminer si une variable est une constante de région simplement en regardant sa définition, si c'est une constante ou si le bloc domine la région de la boucle alors la variable est une constante de région.

Pour déterminer si une variable est une variable d'induction, l'algorithme vérifie la forme des instructions qui mettent à jour la variable, les formes de mises à jours autorisées sont :

1. une variable d'induction plus une constante de région
2. une variable d'induction moins une constante de région
3. une fonction  $\phi$  avec comme arguments une variable d'induction et une constante de région.
4. une copie de registre à registre à partir d'une variable d'induction.

Une variable d'induction est définie par rapport à une certaine boucle. Il peut y avoir des boucles imbriquées, et une variable d'induction pour une boucle peut être une constante de région pour une autre. Une boucle est repérée par une composante fortement connexe du CFG. On parle donc du status d'une variable par rapport à une certaine boucle. On va repérer les boucles, et lancer OSR sur une région particulière de CFG qui correspond à une boucle.

L'algorithme va travailler sur le graphe de dépendance mais pour des raisons pratiques on va l'orienter de manière inhabituelle : il y aura un arc  $i \rightarrow j$  si  $j$  est utilisé pour calculer  $i$  (c'est le graphe qui indique les définitions à partir des utilisations appelé aussi graphe de SSA). Pour notre code, le graphe est le suivant :



Chaque variable d'induction est nécessairement définie dans une composante fortement connexe (CFC) du graphe de SSA, l'algorithme peut identifier toutes les CFC du graphe et tester chaque opération pour vérifier que c'est une opération valide pour la mise à jour d'une variable d'induction la fonction *ClassifyIV* fait cela. L'algorithme complet est le suivant :

OSR( $G$ )

```

nextNum  $\leftarrow$  0
Tant qu'il existe un noeud
   $n$  non visité dans  $G$ 
  DFS( $n$ )

```

DFS( $n$ )

```

 $n.Num$   $\leftarrow$  nextNum ++
 $n.Visited$   $\leftarrow$  true
 $n.Low$   $\leftarrow$   $n.Num$ 
push( $n$ )
pour chaque opérande  $o$  de  $n$ 
  si  $o.Visited = false$  alors
    DFS( $o$ )
     $n.Low$   $\leftarrow$  min( $n.Low, o.Low$ )
  si  $o.Num < n.Num$  et  $o$  est sur la pile
     $n.Low$   $\leftarrow$  min( $n.Low, o.Low$ )
endfor
si  $n.Low = n.Num$  alors
  SCC  $\leftarrow$   $\emptyset$ 
  jusqu'à ce que  $x = n$  faire
     $x$   $\leftarrow$  pop()
    SCC  $\leftarrow$  SCC  $\cup$  { $x$ }
Process(SCC)

```

Process( $r$ )

```

si  $r$  à un seul élément  $n$  alors
  si  $n$  est une opération candidate
    Replace( $n, iv, rc$ )
  sinon  $n.Header$   $\leftarrow$  NULL
sinon
  ClassifyIV( $r$ )

```

ClassifyIV( $r$ )

```

Pour chaque noeud  $n \in r$ 
  si  $n$  n'est pas une mise à jour
    valide pour une variable d'induction
   $r$  n'est pas une variable d'induction
  si  $r$  est une variable d'induction
    header  $\leftarrow$   $n \in r$  avec le plus petit  $n.Low$ 
  Pour chaque noeud  $n \in r$ 
     $n.Header$   $\leftarrow$  header
sinon
  Pour chaque noeud  $n \in r$ 
    si  $n$  est une opération candidate
      Replace( $n, iv, rc$ )
    sinon
       $n.Header$   $\leftarrow$  NULL

```

Pour trouver les CFC dans le graph de SSA, OSR utilise l'algorithme de Tarjan : la fonction DFS. Elle effectue une recherche en profondeur d'abord sur le graphe. Elle assigne à chaque noeud un numéro, qui correspond à l'ordre dans lequel DFS visite les noeuds. Elle pousse chaque noeud sur une pile et étiquette le noeud avec la plus petite étiquette qu'elle peut récupérer de ses enfants (c'est à dire le plus petit numéro lors d'une recherche en profondeur d'abord). Lorsqu'elle a fini de traiter ses enfants, si l'étiquette récupérée à le même numéro que  $n$  alors  $n$  est le noeud d'entrée d'une composante fortement connexe. Tous les noeud de la CFC sont dans la pile.

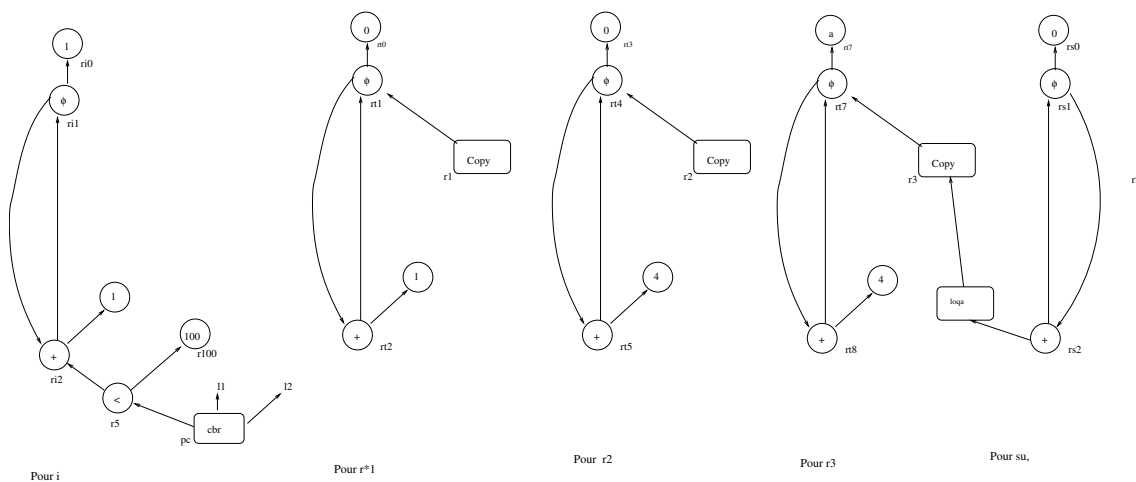
Cette méthode pour trouver les CFC a une propriété intéressante : lorsqu'une CFC est dépilée, DFS a déjà visité tous ses enfants. Les opérations candidates sont donc visitées après que leurs arguments aient été traités. On va donc repérer les variables d'induction avant d'essayer de réduire les opérations candidates qui l'utilisent.

Lorsque DFS trouve une nouvelle CFC, elle est traitée avec *Process*. La procédure *Replace* remplace effectivement une opération candidate et génère le nouveau code.

La fonction *Replace* prend comme argument une opération candidate, et ses arguments : une variable d'induction (*iv*) et une constante de région (*rc*). *Replace* vérifie si la variable d'induction existe déjà (en utilisant une table de hachage indexée par l'opération et les deux arguments). Si elle existe, il remplace l'opération candidate par une copie. Si elle n'existe pas encore, *Replace* crée une nouvelle CFC dans le graphe de SSA, ajoute au graphe de SSA les opérations correspondant au nouveau code, ajoute le nom de la variable dans la table de hachage et remplace l'opération candidate par une copie de la nouvelle variable d'induction. (lors de ce traitement, *Replace* devra tester chaque opération ajoutée pour voir si c'est une opération candidate et éventuellement la réduire).

Sur notre exemple, supposons qu'OSR commence avec le noeud  $r_{s2}$  et qu'il visite les fils gauche avant les fils droit. Il fait des appels récurrents à DFS sur les noeuds  $r_4, r_3, r_2, r_1$  et  $r_{i1}$ . À  $r_{i1}$  il fait des appels récurrents sur  $r_{i2}$  puis  $r_{i0}$ . Il trouve les deux CFC simple contenant la constant 1. Elles ne sont pas candidates. *Process* les marque comme non-variables d'induction en mettant leur champs *header* à Null. La première CFC non triviale découverte est  $\{r_{i1}, r_{i2}\}$ , toutes les opérations sont valides pour une variable d'induction donc *ClassifyIV* marque chaque noeud comme variable d'induction en mettant leur champ *header* qui pointe sur le noeud avec le plus petit numéro de profondeur (ici  $r_{i1}$ ).

Ensuite DFS revient au noeud  $r_1$ , son fils gauche est une variable d'induction et son fils droit est une constante de région. *Replace* est invoquée pour créer une variable d'induction. Cette variable  $r_1$  vaut  $r_{i1} - 1$ , la nouvelle variable d'induction à donc le même incrément que  $r_1$ . On ajoute une copie  $r_1 \leftarrow r_{i1}$  et on marque  $r_1$  comme une variable d'induction. Voici les deux morceaux de graphe de SSA construit à cet instant sont les deux graphe ci-dessous sur la gauche :



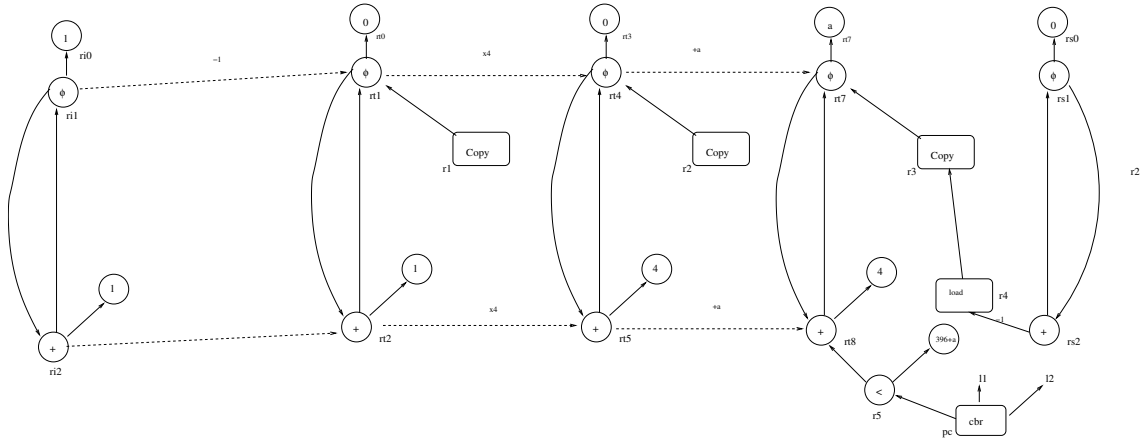
Ensuite, on essaye de réduire  $r_2 \leftarrow r_1 \times 4$ , cette opération est candidate, la procédure *Replace* duplique la variable d'induction  $r_1$ , ajuste l'incrément en fonction de la multiplication et ajoute une copie vers  $r_2$ . L'instruction  $r_3 \leftarrow r_2 + \partial a$  duplique l'induction précédente en ajustant l'initialisation.

Ensuite, DFS arrive sur la CFC qui calcule sum, ce n'est pas une variable d'induction car  $r_4$  n'est pas une constante de région. Restent les noeuds non visités depuis le branchement : il y a une comparaison avec une variable déjà rencontrée. Le graphe de SSA à ce niveau est représenté ci-dessus.

À ce stade on peut repérer que les variables  $r_1$  et  $r_2$  sont morte et la variable  $r_3$  est calculée plus simplement qu'auparavant. On peut en général encore optimiser le résultat en ne conservant, comme utilisation des variables d'inductions originales que le test de fin de boucle. Pour cela, on

effectue un remplacement de fonction de test linéaire (*linear function test replacement*). Le compilateur repère les variables d'inductions qui ne sont pas utilisées dans le test de fin de boucle. Pour cela, le compilateur (1) repère les comparaisons qui sont faites sur des variables d'inductions non utilisées par ailleurs, (2) recherche la nouvelle variable d'induction sur laquelle le test doit être fait, (3) calcule la constante de région nécessaire pour réécrire le test et (4) réécrit le code.

Certaines de ces opérations peuvent être préparées dans OSR, en particulier (1), OSR peut aussi ajouter un arc fictif entre les noeuds qui ont donné lieu à des copies, en étiquetant ces arcs par les transformations apportées. Dans notre exemple, la série de réduction crée une chaîne de noeuds sur lesquels on rencontre les étiquettes  $-1$ ,  $\times 4$  et  $+a$ . Par exemple, voici le graphe de SSA augmenté des arcs fictifs et transformé par remplacement des fonctions de test linéaires.



Voici le code après réécriture :

	<i>loadI</i>	$0$	$\Rightarrow$	$r_{s0}$
	<i>loadI</i>	$@a$	$\Rightarrow$	$r_{t6}$
	<i>addI</i>	$r_{t6}, 396$	$\Rightarrow$	$r_{lim}$
$l_1$	<i>phi</i>	$r_{t6}, r_{t8}$	$\Rightarrow$	$r_{t7}$
	<i>phi</i>	$r_{s0}, r_{s2}$	$\Rightarrow$	$r_{s1}$
	<i>load</i>	$r_{t7}$	$\Rightarrow$	$r_4$
	<i>add</i>	$r_4, r_{s1}$	$\Rightarrow$	$r_{s2}$
	<i>addI</i>	$r_{t7}, 4$	$\Rightarrow$	$r_{t8}$
	<i>cmp_LE</i>	$r_{t7}, r_{lim}$	$\Rightarrow$	$r_5$
	<i>cbr</i>	$r_5$	$\rightarrow$	$l_1, l_2$
$l_2$	$\dots$			

## A Opération de l'assembleur linéaire Iloc

La syntaxe générale est *opCode Sources*  $\Rightarrow$  *Target*

Ci-dessous,  $r_i$  représente un registre,  $c_i$  une constante  $l_i$  un label

<i>Opcode</i>	<i>Sources</i>	<i>cible</i>	<i>Semantique</i>
<i>nop</i>	--	--	ne fait rien
<i>add</i>	$r_1, r_2$	$r_3$	$r_1 + r_2 \rightarrow r_3$
<i>addI</i>	$r_1, c_1$	$r_2$	$r_1 + c_1 \rightarrow r_2$
<i>sub</i>	$r_1, r_2$	$r_3$	$r_1 - r_2 \rightarrow r_3$
<i>subI</i>	$r_1, c_1$	$r_2$	$r_1 - c_1 \rightarrow r_2$
<i>mult</i>	$r_1, r_2$	$r_3$	$r_1 \times r_2 \rightarrow r_3$
<i>multI</i>	$r_1, c_1$	$r_2$	$r_1 \times c_1 \rightarrow r_2$
<i>div</i>	$r_1, r_2$	$r_3$	$r_1 \div r_2 \rightarrow r_3$
<i>divI</i>	$r_1, c_1$	$r_2$	$r_1 \div c_1 \rightarrow r_2$
<i>lshift</i>	$r_1, r_2$	$r_3$	$r_1 \ll r_2 \rightarrow r_3$
<i>lsiftI</i>	$r_1, c_1$	$r_2$	$r_1 \ll c_1 \rightarrow r_2$
<i>rshift</i>	$r_1, r_2$	$r_3$	$r_1 \gg r_2 \rightarrow r_3$
<i>rshiftI</i>	$r_1, c_1$	$r_2$	$r_1 \gg c_1 \rightarrow r_2$
<i>and</i>	$r_1, r_2$	$r_3$	$r_1 \wedge r_2 \rightarrow r_3$
<i>andI</i>	$r_1, c_1$	$r_2$	$r_1 \wedge c_1 \rightarrow r_2$
<i>or</i>	$r_1, r_2$	$r_3$	$r_1 \vee r_2 \rightarrow r_3$
<i>orI</i>	$r_1, c_1$	$r_2$	$r_1 \vee c_1 \rightarrow r_2$
<i>xor</i>	$r_1, r_2$	$r_3$	$r_1 \text{xor } r_2 \rightarrow r_3$
<i>xorI</i>	$r_1, c_1$	$r_2$	$r_1 \text{xor } c_1 \rightarrow r_2$
<i>loadI</i>	$c_1$	$r_2$	$c_1 \rightarrow r_2$
<i>load</i>	$r_1$	$r_2$	$Memory(r_1) \rightarrow r_2$
<i>loadAI</i>	$r_1, c_1$	$r_2$	$Memory(r_1 + c_1) \rightarrow r_2$
<i>loadAO</i>	$r_1, r_2$	$r_3$	$Memory(r_1 + r_2) \rightarrow r_3$
<i>cload</i>	$r_1$	$r_2$	<i>load</i> pour caractères
<i>cloadAI</i>	$r_1, c_1$	$r_2$	<i>loadAI</i> pour caractères
<i>cloadAO</i>	$r_1, r_2$	$r_3$	<i>loadAO</i> pour caractères
<i>store</i>	$r_1$	$r_2$	$r_1 \rightarrow Memory(r_2)$
<i>storeAI</i>	$r_1$	$r_2, c_1$	$r_1 \rightarrow Memory(r_2 + c_1)$
<i>storeAO</i>	$r_1$	$r_2, r_3$	$r_1 \rightarrow Memory(r_2 + r_3)$
<i>cstore</i>	$r_1$	$r_2$	<i>store</i> pour caractères
<i>cstoreAI</i>	$r_1$	$r_2, c_1$	<i>storeAI</i> pour caractères
<i>cstoreAO</i>	$r_1$	$r_2, r_3$	<i>storeAO</i> pour caractères
<i>i2i</i>	$r_1$	$r_2$	$r_1 \rightarrow r_2$
<i>c2c</i>	$r_1$	$r_2$	$r_1 \rightarrow r_2$
<i>c2i</i>	$r_1$	$r_2$	convertit un caractère en entier
<i>i2c</i>	$r_1$	$r_2$	convertit un entier en caractère

Pour le contrôle de flot :

<i>Opcode</i>	<i>Sources</i>	<i>cible</i>	<i>Semantique</i>
<i>cbr</i>	$r_1$	$l_1, l_2$	$r_1 = True \Rightarrow l_1 \rightarrow PC, r_1 = False \Rightarrow l_2 \rightarrow PC$
<i>jumpI</i>	--	$l_1$	$l_1 \rightarrow PC$
<i>jump</i>	--	$r_1$	$r_1 \rightarrow PC$
<i>cmp_LT</i>	$r_1, r_2$	$r_3$	$r_1 < r_2 \Rightarrow True \rightarrow r_3, r_1 \geq r_2 \Rightarrow False \rightarrow r_3$
<i>cmp_LE</i>	$r_1, r_2$	$r_3$	$r_1 \leq r_2 \Rightarrow True \rightarrow r_3$
<i>cmp_EQ</i>	$r_1, r_2$	$r_3$	$r_1 = r_2 \Rightarrow True \rightarrow r_3$
<i>cmp_NE</i>	$r_1, r_2$	$r_3$	$r_1 \neq r_2 \Rightarrow True \rightarrow r_3$
<i>cmp_GE</i>	$r_1, r_2$	$r_3$	$r_1 \geq r_2 \Rightarrow True \rightarrow r_3$
<i>cmp_GT</i>	$r_1, r_2$	$r_3$	$r_1 > r_2 \Rightarrow True \rightarrow r_3$
<i>tbl</i>	$r_1, l_2$	--	$r_1$ peut contenir $l_1$

Autre syntaxe pour les branchements

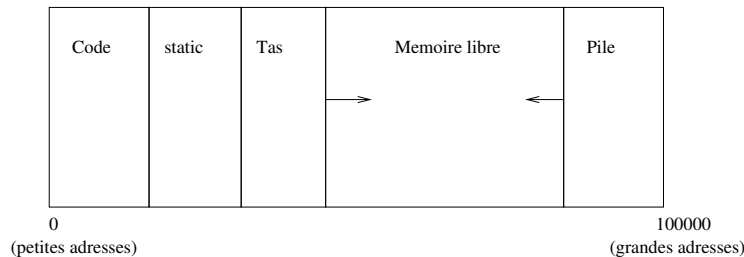
<i>Opcode</i>	<i>Sources</i>	<i>cible</i>	<i>Semantique</i>
<i>comp</i>	$r_1, r_2$	$cc_1$	definit $cc_1$
<i>cbr_LT</i>	$cc_1$	$l_1, l_2$	$cc_1 = LT \Rightarrow l_1 \rightarrow PC, cc_1 \neq LT \Rightarrow l_2 \rightarrow PC,$
<i>cbr_LE</i>	$cc_1$	$l_1, l_2$	$cc_1 = LE \Rightarrow l_1 \rightarrow PC$
<i>cbr_EQ</i>	$cc_1$	$l_1, l_2$	$cc_1 = EQ \Rightarrow l_1 \rightarrow PC$
<i>cbr_GE</i>	$cc_1$	$l_1, l_2$	$cc_1 = GE \Rightarrow l_1 \rightarrow PC$
<i>cbr_GT</i>	$cc_1$	$l_1, l_2$	$cc_1 = GT \Rightarrow l_1 \rightarrow PC$
<i>cbr_NE</i>	$cc_1$	$l_1, l_2$	$cc_1 = NE \Rightarrow l_1 \rightarrow PC$



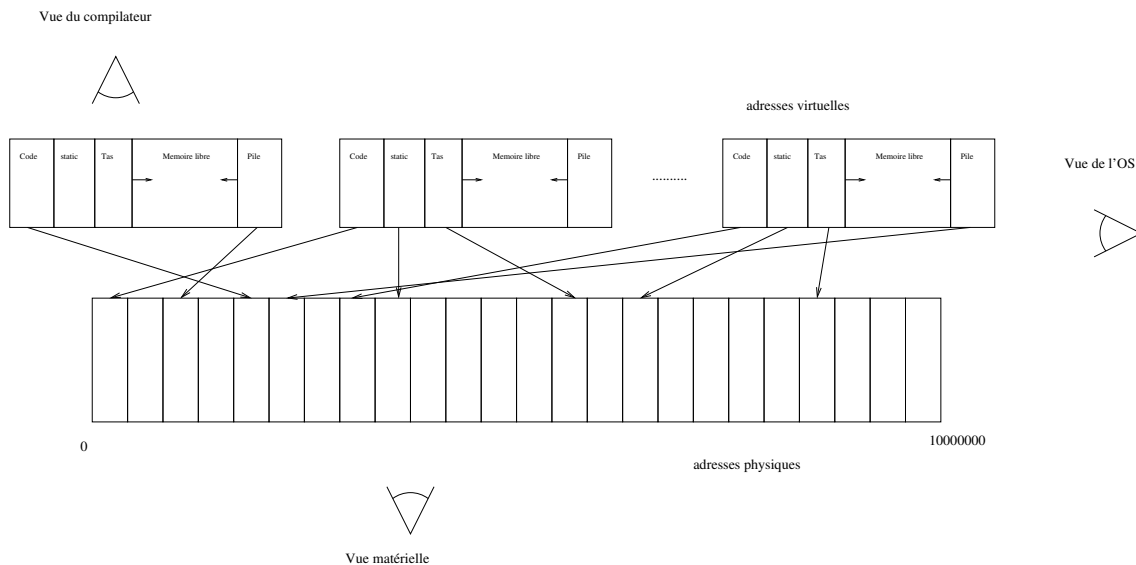
## B Gestion de la mémoire globale

Chaque programme comporte son propre adressage logique. L'exécution de plusieurs proces- sus en parallèle (découpage du temps) implique des règles strictes pour la gestion mémoire.

**Organisation physique de la mémoire** On a vu que le compilateur voyait la mémoire de la manière suivante :



En réalité, le système d'exploitation assemble plusieurs espaces d'adressage virtuels ensemble dans la mémoire physique. Les différentes mémoires virtuelles sont découpées en pages qui sont arrangées dans la mémoire physique de manière complètement incontrôlable. Cette correspon- dance est maintenue conjointement par le matériel et le système d'exploitation et est en principe transparente pour le concepteur de compilateur.



Cette correspondance est nécessaire car l'espace adressable en adresse virtuelle est beaucoup plus grand que l'espace mémoire disponible en mémoire vive. La mémoire de la machine fonc- tionne alors comme un gros cache : lorsqu'il n'y a plus de place, on *swappe* un page, c'est à dire qu'on la copie sur le disque (en fait c'est un cache à deux niveaux car il y a aussi le cache de donnée et d'instruction bien sûr). L'adressage d'une case mémoire depuis un programme comporte donc les opérations suivantes :

- traduction de l'adresse virtuelle en adresse physique
- Vérification que la page correspondant à l'adresse physique est en mémoire et (éventuellement) génération d'un défaut de page (*page fault*)
- lire (ou écrire) la donnée effectivement. Comme tout système de cache, l'écriture nécessite quelques précautions.

En général la page des tables (page indiquant la correspondance entre pages virtuelles et pages physiques) se trouve elle-même en mémoire. Pour accélérer le décodage, il existe souvent un composant matériel spécifique, dédié à la traduction entre les adresses virtuelles et les adresses physiques (*memory management unit*, MMU). Aujourd'hui, sur la majorité des systèmes, la MMU

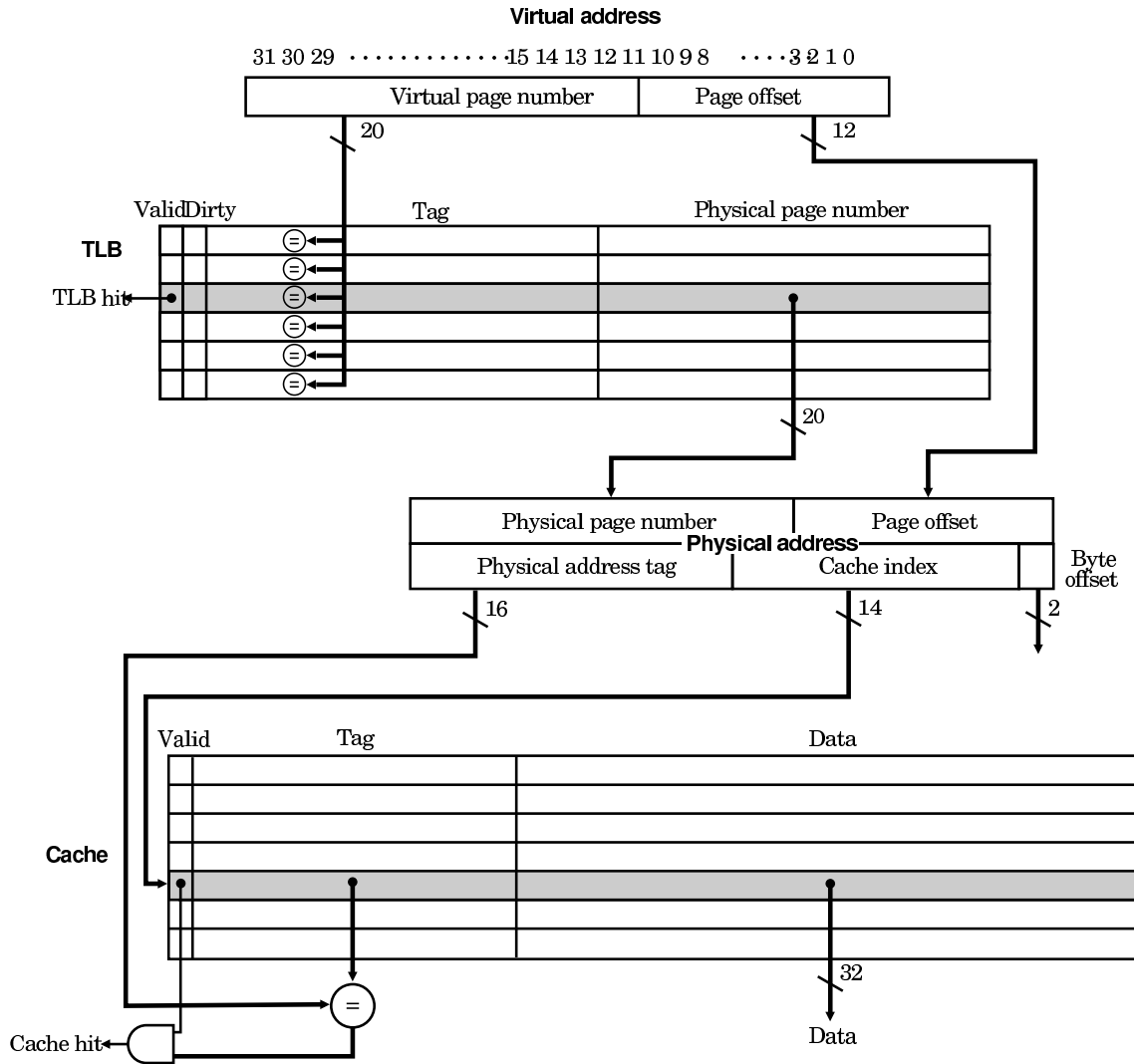


FIG. 2 – Implémentation du TLB et du cache sur les DECStation 3100 (figure tirée de [HP98]).

consiste en un *translation lookahead buffer* (TLB), c'est un petit cache contenant la correspondance entre page virtuelle et page physique pour les pages les plus récemment utilisées.

Voici quelques valeurs typiques pour un système mémoire sur un ordinateur :

- TLB
  - Taille du TLB : entre 32 et 4096 entrées
  - Pénalité d'un défaut de TLB 10-100 cycles
  - Taux de défauts de TLB : 0.01%-1%
- cache
  - Taille du cache 8 à 16 000 Koctets en blocs de 4 à 256 octets
  - Pénalité d'un défaut de cache : 10 à 100 cycles
  - Taux de défauts de cache : 0.1%-10%
- mémoire paginée
  - Taille de la mémoire : 8 à 256 000 Koctets organisée en pages de 4 à 256 Koctets
  - Pénalité d'un défaut de page : entre 1 et 10 millions de cycles
  - Taux de défauts de page :  $10^{-5}\%$ - $10^{-4}\%$ .

La figure 2, tirée du Hennessy-Patterson [HP98] présente l'organisation du système mémoire sur les DECStation 3100.

## Organisation de la mémoire virtuelle

- Alignement et *padding*. Les machines cibles ont des restrictions quant à l'alignement des mots en mémoire ; par exemple les entiers doivent commencer à une césure de mot (32 bits), les longs mots doivent commencer à une césure long mot (64 bits), ce sont les règles d'alignement (*alignment rules*). Pour respecter ces contraintes, le compilateur classe ses variables en groupes du plus restrictif (du point de vue des restrictions liées à l'alignement) au moins restrictif. Lorsque qu'il n'est pas possible de respecter les règles d'alignement sans faire de trou, le compilateur insère des trous (*padding*).
- Gestion des caches. À ce niveau, il est important de prendre en compte que toutes les machines possèdent un cache. Les performances liées au cache peuvent être améliorées en augmentant la *localité*. Par exemple, deux variables susceptibles d'être utilisées en même temps vont partager le même bloc de cache. La lecture de l'une des deux rapatriera l'autre dans le cache à la même occasion. Si cela n'est pas possible, le compilateur peut essayer de s'assurer que ces deux données sont dans des blocs qui ne vont pas être mis sur la même ligne de cache.

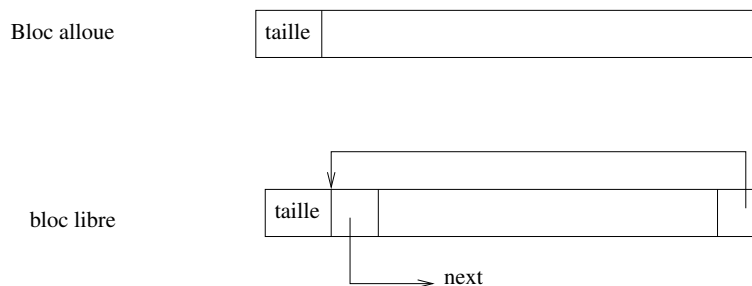
En général, le problème est très complexe. Le compilateur va se focaliser sur le corps des boucles et il peut choisir d'ajouter des trous entre certaines variables pour diminuer leur chance d'être en conflit sur le cache.

Cela marche bien lorsque le cache fonctionne avec des adresses virtuelles (*virtual adress cache*), c'est à dire que le cache recopie des morceaux de mémoire virtuelle (et donc l'écart des données en mémoire virtuelle influe vraiment sur leur présence simultanée dans le cache). Si le cache fonctionne avec la mémoire physique (*physical adress cache*), la distance entre deux données n'est contrôlable que si les deux données sont dans la même page, le compilateur s'intéresse alors aux données référencées en même temps sur la même page.

**gestion du tas** La gestion du tas doit assurer (dans l'ordre d'importance)

1. l'appel (ou pas) explicite à la routine de libération de mémoire,
2. des routines efficaces d'allocation et de libération de mémoire,
3. la limitation de la fragmentation du tas en petit blocs.

La méthode traditionnelle est appelée allocation du premier (*first-fit allocation*). Chaque bloc alloué a un champ caché qui contient sa taille (en général juste avant le champ retourné par la procédure d'allocation). Les blocs disponibles pour allocation sont accessibles par une liste dite liste libre (*free list*), chaque bloc sur cette liste contient un pointeur vers le prochain bloc libre dans la liste.

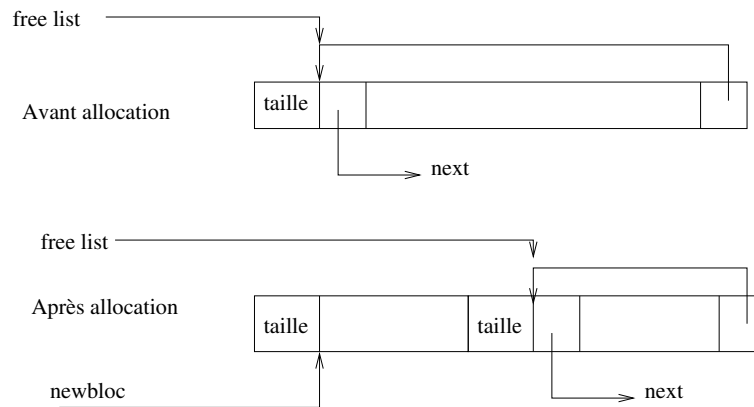


L'allocation d'un bloc se fait de la manière suivante (il démarre avec un unique bloc libre contenant toute la mémoire allouée au tas) :

- parcours de la liste libre jusqu'à la découverte d'un bloc libre de taille suffisante.
- si le bloc est trop grand, création d'un nouveau bloc libre et ajout à la liste libre.
- la procédure retourne un pointeur sur le bloc choisi.
- si la procédure ne trouve pas de bloc suffisamment grand, elle essaie d'augmenter le tas. Si elle y arrive, elle retourne un bloc de la taille voulue.

Pour libérer un bloc, la procédure la plus simple est d'ajouter le bloc à la liste libre. Cela produit une procédure de libération rapide et simple. Mais cela conduit en général à fragmenter trop vite la mémoire. Pour essayer d'agglomérer les blocs, l'allocateur peut tester le mot précédent le

champ de taille du bloc qu'il libère. Si c'est un pointeur valide (i.e. si le bloc précédent est dans la liste libre), on ajoute le nouveau bloc libre à celui-ci en augmentant sa taille et mettant à jour le pointeur de fin. Pour mettre à jour efficacement la liste libre, il est utile qu'elle soit doublement chaînée.



Les allocateurs modernes prennent en compte le fait que la taille de mémoire est de moins en moins un paramètre critique alors que la vitesse d'accès à la mémoire l'est de plus en plus. De plus, on remarque que l'on alloue fréquemment des petites tailles standard et peu fréquemment des grandes tailles non standard. On utilise alors des zones mémoires différentes pour chaque taille, commençant à 16 octets allant par puissance de deux jusqu'à la taille d'une page (2048 ou 4096 octets). On alloue des blocs d'une taille redéfinie. La libération n'essaie pas d'agrèger des blocs

**libérations mémoire implicites** Beaucoup de langages spécifient que l'implémentation doit libérer la mémoire lorsque les objets ne sont plus utilisés. Il faut donc inclure un mécanisme pour déterminer quand un objet est mort. Les deux techniques classiques sont le comptage des références (*reference counting*) et le ramasse-miettes (*garbage collecting*).

Le comptage des références augmente chaque objet d'un compteur qui compte le nombre de pointeurs qui référencent cet objet. Chaque assignation à un pointeur modifie deux compteurs de référence (celui de l'ancienne valeur pointée et celui de la nouvelle valeur pointée). Lorsque le compteur d'un objet tombe à zéro, il peut être ajouté à la liste libre (il faut alors enregistrer le fait que tous les pointeurs de cet objet sont désactivés).

Ce mécanisme pose plusieurs problèmes :

- Le code s'exécutant a besoin d'un mécanisme permettant de distinguer les pointeurs des autres données. On peut étendre encore les objets pour indiquer quel champ de chaque objet est un pointeur ou diminuer la zone utilisable d'un pointeur et "tagger" chaque pointeur.
- Le travail à faire pour une simple libération de pointeur peut être arbitrairement gros. Dans les systèmes soumis à des contraintes temps-réel, cela peut être un problème important. On emploie alors un protocole plus complexe qui limite la quantité d'objets libérés en une fois.
- Un programme peut former des graphes cycliques de pointeurs. On ne peut pas mettre à zéro les compteurs des objets sur un circuit, même si rien ne pointe sur les circuits. Le programmeur doit casser les circuits avant de libérer le dernier pointeur qui pointe sur ces circuits.

La technique du ramasse-miettes ne libère rien avant de ne plus avoir de place en mémoire. Quand cela arrive, on arrête l'exécution et on scanne la mémoire pour traquer les pointeurs pour découvrir les zones mémoire inoccupées. On libère alors la mémoire en compactant (ou pas) la mémoire à la même occasion.

La première phase détermine quel ensemble d'objets peut être atteint à partir des pointeurs des variables du programme et des temporaires générés par le compilateur. Les objets non accessibles par cet ensemble sont considérés comme morts. La deuxième phase libère explicitement ces objets et les recycle. Les deux techniques de ramasse-miettes (*mark-sweep* et *copying collectors*) diffèrent par cette deuxième phase de recyclage.

La première phase procède par marquage des objets, chaque objet possède un bit de marquage (*mark bit*). On commence par mettre tous ses drapeaux à 0 et on initialise une liste de travail qui contient tous les pointeurs contenus dans les registres et dans les AR des procédures en cours. Ensuite on descend le long des pointeurs et l'on marque chaque objet rencontré, en recommençant avec les nouveaux pointeurs. Exemple d'algorithme :

```
Initialiser les marques à 0
WorkList ← { valeurs des pointeurs des AR et registres }
While (WorkList ≠ 0)
    p ← tete de WorkList
    if (p → objet non marqué)
        marquer p → objet
        ajouter les pointeurs de p → objet à WorkList
```

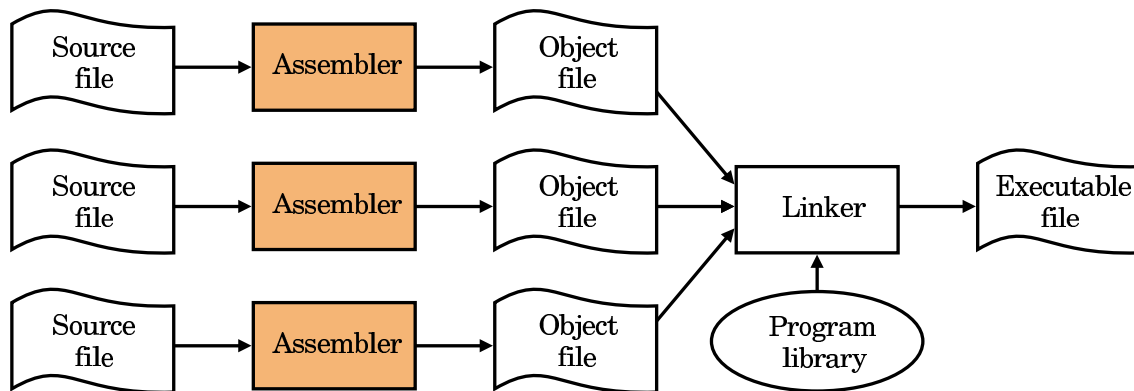
La qualité de l'algorithme (précis ou approximatif) vient de la technique utilisée pour identifier les pointeurs dans les objets. Le type de chaque objet peut éventuellement être enregistré dans l'objet lui-même (*header*), dans ce cas on peut déterminer exactement les pointeurs. Sinon tous les champs d'un objet sont considérés comme des pointeurs. Ce type de ramasse-miettes peut être utilisé pour des langages qui n'ont habituellement pas de ramasse-miettes (ex : C).

Les ramasse-miettes du type *mark-sweep* font une simple passe linéaire sur le tas et ajoutent les objets morts à la liste libre comme l'allocateur standard. Les ramasse-miettes du type *copy-collector* travaillent avec un tas divisé en deux parties de taille égale (old et new) ; lorsqu'il n'y a plus de place dans old, ils copient toutes les données vivantes (et donc les compactent) de old dans new et échangent simplement les noms old et new (la copie peut être faite après la première phase ou incrémentalement au fur et à mesure de la première phase).

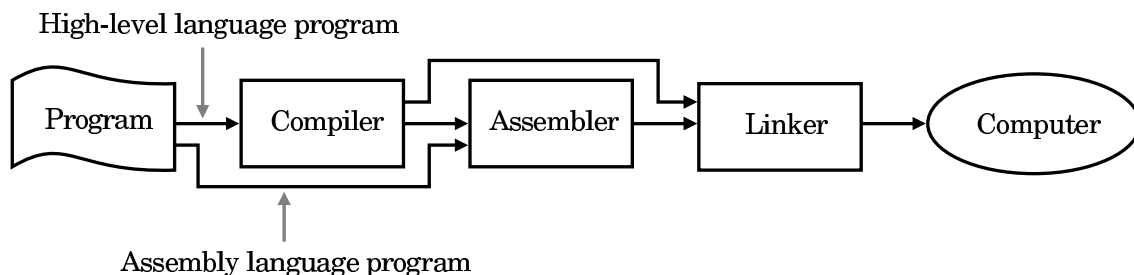
## C Assembleur, éditeur de liens, l'exemple du Mips

Le processus de compilation ne produit pas directement un code exécutable, il reste les processus d'assemblage, d'édition de lien et de chargement de l'exécutable qui peuvent modifier le programme. Ces phases ne sont plus uniquement dépendantes du jeu d'instruction de la machine, elles dépendent des conventions utilisées pour le format de l'exécutable, des tâches réalisées par le système d'exploitation ainsi que du matériel présent sur le circuit (unité de gestion de la mémoire par exemple, *memory management unit*, MMU). Cette annexe décrit ces étapes pour le processeur Mips, elle est tirée de l'annexe du Hennessy-Patterson [HP98], écrit par J. R. Larus ([http://www.cs.wisc.edu/~larus/SPIM/HP\\_AppA.pdf](http://www.cs.wisc.edu/~larus/SPIM/HP_AppA.pdf)). Il existe simulateur du jeu d'instruction du Mips (Instruction Set Simulateur : ISS) appelé Spim qui peut être téléchargé à l'adresse : <http://www.cs.wisc.edu/~larus/spim.html> (voir le figure 4 plus loin). Ce type de simulateur est utilisé dès que l'on développe des portions critiques de logiciel (performance, pilote de périphérique, etc.).

Le langage *assembleur*<sup>3</sup> est une représentation symbolique du langage machine (binaire) de la machine. Il permet en particulier l'utilisation d'étiquette (*labels*<sup>4</sup>) pour désigner des emplacements mémoire particuliers. L'outil appelé *assembleur* permet de traduire le langage assembleur vers le langage machine. Lors de la compilation d'un programme, plusieurs sources assembleur sont compilés indépendamment vers des programmes objets et des bibliothèques pré-compilées sont aussi utilisées. Un autre outil, l'éditeur de lien, combine tous les programmes objets en un programme exécutable. Le schéma global est donc le suivant :



Les compilateurs usuels incluent directement l'assembleur dans le traitement effectué sur le langage de haut niveau, mais plusieurs chemin de compilation sont possibles :



Lorsque les performances du programme exécutable sont critiques (par exemple pour les systèmes embarqués), il est encore d'usage de programmer en assembleur, ou au moins de vérifier que l'assembleur généré par le compilateur n'est pas trop inefficace (on peut généralement restreindre la partie traitée en assembleur à une petite partie critique du programme global). Pour cette raison, il est important de bien connaître le langage assembleur ainsi que les processus travaillant dessus (assembleur et éditeur de liens).

<sup>3</sup>Attention, en français on utilise le même terme "assembleur" pour désigner le langage (assembleur ou langage d'assemblage) et l'outil qui compile le langage (assembleur)

<sup>4</sup>j'utiliserai le terme *labels* dans la suite

```

                                .text
                                .align 2
                                .globl main
main:
    subu    $sp, $sp, 32
    sw     $ra, 20($sp)
    sd     $a0, 32($sp)
    sw     $0, 24($sp)
    sw     $0, 28($sp)
#include <stdio.h>
int
main (int argc, char *argv[])
{
int i;
int sum = 0;
for (i = 0; i <= 100; i = i + 1)
    sum = sum + i * i;
printf ("The sum from 0 .. \
    100 is %d\n", sum);
}
                                loop:
                                lw     $t6, 28($sp)
                                mul    $t7, $t6, $t6
                                lw     $t8, 24($sp)
                                addu   $t9, $t8, $t7
                                sw     $t9, 24($sp)
                                addu   $t0, $t6, 1
                                sw     $t0, 28($sp)
                                ble    $t0, 100, loop
                                la     $a0, str
                                lw     $a1, 24($sp)
                                jal    printf
                                move   $v0, $0
                                lw     $ra, 20($sp)
                                addu   $sp, $sp, 32
                                jr     $ra
                                .data
                                .align 0
str:
                                .asciiz "The sum from 0 .. 100 is %d\n"

```

FIG. 3 – Un exemple de programme C et de code assembleur Mips correspondant

## C.1 Assembleur

Un exemple de programme assembleur Mips, ainsi que le programme C correspondant, est montré en figure 3. Les noms commençant par un point sont des directives (aussi appelées balises) qui indiquent à l'assembleur comment traduire un programme mais ne produisent pas d'instruction machine directement. Les nom suivit de deux points (`main`, `str`) sont des labels. La directive `.text` indique que les lignes qui suivent contiennent des instructions. La directive `.data` indique que les lignes qui suivent contiennent des données. La directive `.align n` indique que les objets qui suivent doivent être alignés sur une adresse multiple de  $2^n$  octets. Le label `main` est déclaré comme global. La directive `.asciiz` permet de définir une chaîne de caractères terminée par le caractère `null`.

Le programme assembleur traduit un fichier de format assembleur en un fichier contenant les instructions machine et les données au format binaire (format *objet*). Cette traduction se fait en deux passes : la première passe repère les labels définis, la deuxième passe traduit chaque instruction assembleur en instruction binaire légale.

Le fichier résultant est appelé un *fichier objet*. Un fichier objet n'est en général pas exécutable car il contient des références à des labels *externes* (appelé aussi *global*, ou *non-local*). Un label est *local* si il ne peut être référencé que depuis le fichier ou il est défini. Dans la plupart des assembleurs, les labels sont locaux par défaut, à moins d'être explicitement spécifié comme globaux. Le mot clés `static` en C force l'objet à n'être visible que dans le fichier ou il est défini. L'assembleur produit donc une liste de labels qui n'ont pu être *résolus* (i.e. dont on ne trouve pas l'adresse dans le fichier ou ils sont définis).

Le format d'un fichier objet sous unix est organisé en 6 sections :

Object file header	Text segment	Data segment	Relocation information	Symbol table	Debugging information
--------------------	--------------	--------------	------------------------	--------------	-----------------------

- Le *header* décrit la taille et la position des autres sections dans le fichier.
- La section de texte (ou segment de texte) contient la représentation binaire des instructions contenues dans le fichier. Ces instructions peuvent ne pas être exécutables à cause des références non résolues
- La section de données (ou segment de données) contient la représentation binaire des données utilisées dans le fichier source.
- L'information de relocalisation (ou de relocation) identifie les instructions ou données qui référencent des adresses absolues. Ces références doivent être changées lorsque certaines parties du code sont déplacées dans la mémoire.
- la table des symboles associe une adresse à chaque label global défini dans le fichier et liste les références non résolues à des labels externes.
- les informations de débogage contiennent quelques renseignements sur la manière avec laquelle le programme a été compilé, de manière à ce que le debugger puisse retrouver à quelle ligne du fichier source correspond l'instruction courante.

En l'absence de la connaissance de l'adresse où sera placé le code objet, l'assembleur suppose que chaque fichier assemblé commence à une adresse particulière (par exemple l'adresse 0) et laisse à l'éditeur de lien le soin de *reloger* les adresses en fonctions de l'adresse de l'exécutable final.

En général, l'assembleur permet un certain nombre de facilités utilisées fréquemment :

- Les *directives* permettant de décrire des données de manière plus conviviale qu'en binaire, par exemple :  

```
.ascii "la somme est....%n"
```
- Les *macros* permettent de réutiliser une séquence d'instructions à plusieurs endroits sans la répéter dans le texte.
- Les *pseudo-instructions* sont des instructions proposées par l'assembleur mais non implémentées dans le matériel, c'est le programme assembleur qui traduit ses instructions en instructions exécutables par l'architecture.

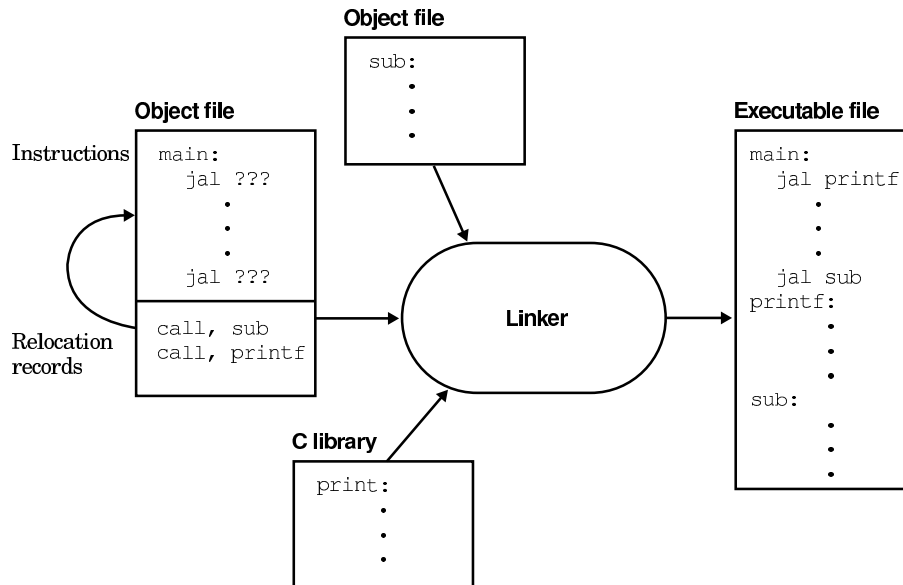
## C.2 Édition de lien

La compilation séparée permet à un programme d'être décomposée en plusieurs parties qui vont être stockées dans différents fichiers. Un fichier peut être compilé et assemblé indépendamment des autres. Ce mécanisme est autorisé grâce à l'édition de lien qui suit l'assemblage. L'éditeur de lien (*linker*) effectue trois tâches :

- Il recherche dans le programme les bibliothèques (*libraries*) utilisées, c'est à dire les appels à des fonctions déjà compilées ailleurs.
- Il détermine les emplacements mémoire que vont occuper les différents fichiers composant le programme et modifie le code de chaque fichier en fonction de ces adresses en ajustant les références absolues
- Il résout les références entre les fichiers.

Un programme qui utilise un label qui ne peut être résolu par l'éditeur de lien après le traitement de tous les fichiers du programme et les bibliothèques disponibles n'est pas correct. Lorsqu'une fonction définie dans une bibliothèque est utilisée, l'éditeur de lien extrait son code de la bibliothèque et l'intègre dans le segment de texte du programme compilé. Cette procédure peut à son tour en appeler d'autres et le processus est reproduit. Voici un exemple simple :





### C.3 Chargement

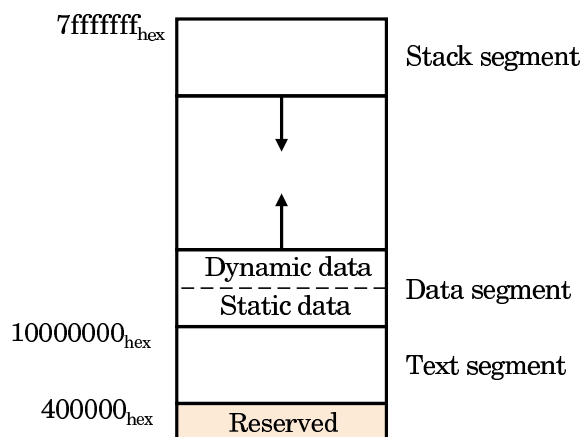
Une fois l'édition de lien réussie, le programme exécutable est stocké sur un système de stockage secondaire, tel qu'un disque dur. Lors de l'exécution sur un système de type Unix, le système d'exploitation charge le programme dans la mémoire vive avant de l'exécuter. Le système effectue donc les opérations suivantes :

1. Lire le header de l'exécutable pour déterminer la taille des segments de texte et de donnée.
2. Créer un nouvel espace d'adressage pour le programme, cet espace est assez grand pour contenir les segments de texte, de donnée ainsi que la pile.
3. Copier les instructions et données de l'exécutable dans le nouvel espace d'adressage.
4. Copier les arguments passés au programme sur la pile.
5. initialiser les registres de la machine. En général les registres sont remis à 0, sauf le pointeur de pile (SP) et le compteur d'instruction (PC).
6. Exécuter une procédure de lancement de programme qui copie les argument du programme dans certains registres et qui appelle la procédure `main` du programme. Lorsque le `main` se termine, la procédure de lancement termine avec l'appel système `exit`.

### C.4 Organisation de la mémoire

La plupart des architectures matérielles de processeurs n'imposent pas de convention sur l'utilisation de la mémoire ou le protocole d'appel de procédure. Ces conventions sont imposées par le système d'exploitation, elles représentent un accord entre les différents programmeurs pour suivre le même ensemble de règles de manière à pouvoir partager les développements réalisés. Nous détaillons ici les conventions utilisées pour les architectures basées sur le Mips.

Sur un processeur Mips, la mémoire vue par le programme est divisée en trois parties. Pour cette vision de la mémoire, on parle encore de mémoire virtuelle :



La première partie, commençant à l'adresse  $400000_{hex}$  est le segment de texte qui contient les instructions du programme. La deuxième partie, au dessus du segment de texte est le segment de données. Il est lui même découpé en deux parties : les données statiques, commençant à l'adresse  $1000000_{hex}$  et les données dynamiques. Les données statiques contiennent toutes les variables déclarées statiquement dans le programme dont on connaît la taille à la compilation, ces données ont donc une adresse absolue dans l'espace d'adressage du programme. Les données dynamiques sont utilisées pour créer des variables dynamiquement (avec la fonction `C malloc` par exemple). La partie données dynamique peut être agrandie dynamiquement (lors de l'exécution du programme) par le système d'exploitation grâce à l'appel système `sbrk` (sur les systèmes Unix). La troisième partie, le segment de pile, commence à l'adresse  $7fffffff_{hex}$ , c'est à dire à la fin de l'espace adressable et progresse dans le sens des adresses décroissantes. Il est aussi étendu dynamiquement par le système au fur et à mesure que le programme fait descendre le pointeur de pile.

**Remarque :** Du fait que les segments de données commencent à l'adresse  $1000000_{hex}$ , on ne peut pas accéder à ces données avec une adresse de 16 bits. Le jeu d'instruction Mips propose un certain nombre de mode d'adressage utilisant un décalage codé sur 16 bits. Par convention, Pour éviter d'avoir à décomposer chaque chargement en deux instructions, le système dédie un registre pour pointer sur le début du segment de données : Le *global pointeur* (`$gp`) contient l'adresse  $10008000_{hex}$

## C.5 Conventions pour l'appel de procédure

La convention logicielle pour l'appel de procédure du Mips est très proche de celle que nous avons expliquée en section 6, GCC ajoute quelques conventions sur l'utilisation des registres, elle doivent donc être utilisées si l'on veut produire du code compatible avec celui de GCC. Le CPU du Mips contient 32 registres à usage général qui sont numérotés de 0 à 31. La notation `$r` permet de désigner un registre par autre chose qu'un numéro.

- Le registre `$0` contient systématiquement la valeur 0.
- Les registres `$at` (registre numéro 1), `$k0` (reg. 26) et `$k1` (reg. 27) sont réservés pour le système d'exploitation.
- Les registre `$a0-$a3` (reg. 4-7) sont utilisés pour passer les quatre premiers arguments des procédures (les arguments suivants sont passés sur la pile). Les registres `$v0` et `$v1` sont utilisés pour stocker les résultats des fonctions.
- Les registres `$t0-$t9` (reg. 8-15,24,25) sont des registres caller-saved (temporaires), qui sont écrasés lors d'appel de procédure.
- Les registres `$s0-$s9` (reg. 16-23) sont des registres callee-saved qui sont conservés à travers l'appel de procédure.
- Le registre `$gp` (reg. 28) est le pointeur global qui pointe sur le milieu d'un block de 64K dans le segment de données statiques de la mémoire.
- Le registre `$sp`, (reg. 29, *stack pointer*) est le pointeur de pile. Le registre `$fp` (reg. 30, *frame pointer*) est l'ARP. L'instruction `jal` (jump and link) écrit dans le registre `$ra` (reg. 31) l'adresse de retour pour un appel de procédure.

Voici la séparation des tâches (*calling convention*) sur la plupart des machines Mips :

- La procédure appelante effectue :
  1. Passer les arguments : les 4 premiers sont passés dans les registres  $\$a0-\$a4$ , les autres arguments sont empilés sur la pile.
  2. Sauvegarder les registres caller-save  $\$t0-\$t9$  si cela est nécessaire (c.a.d. si l'on veut conserver les valeurs de ces registres après l'appel de la procédure).
  3. Exécuter l'instruction `jal` qui saute à l'adresse de la procédure appelée et copier l'adresse de retour dans le registre  $\$ra$ .
- La procédure appelée effectue :
  1. Allouer la taille de l'enregistrement d'activation sur la pile en soustrayant la taille au registre  $\$fp$ .
  2. Sauvegarder les registres callee-saved  $\$s0-\$s9$ ,  $\$fp$  et  $\$ra$  dans l'enregistrement d'activation.
  3. Établir la valeur du nouvel ARP dans le registre  $\$fp$ .
- Lorsqu'elle a fini son exécution, la procédure appelée effectue les opérations suivantes :
  1. Si la fonction retourne une valeur, placer cette valeur dans le registre  $\$v0$ .
  2. Restaurer les registres callee-saved.
  3. Dépiler l'enregistrement d'activation en ajoutant la taille de l'enregistrement au registre  $\$sp$ .
  4. Retourner, en sautant (`jr`) à l'adresse contenue dans le registre  $\$ra$ .

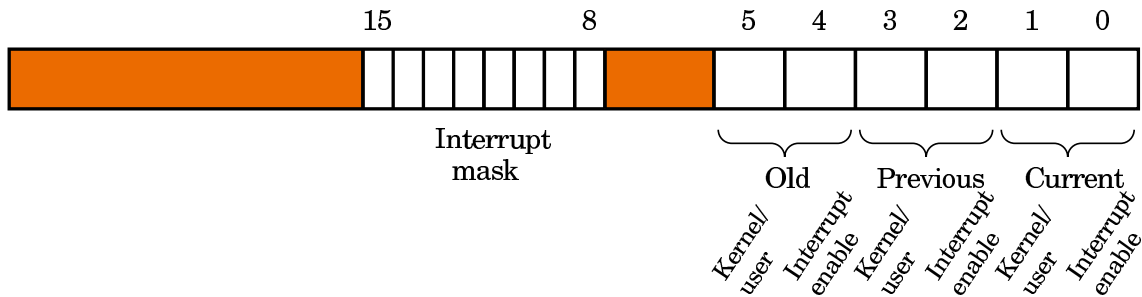
## C.6 Exceptions et Interruptions

En général on appelle exception ce qui vient du processeur (e.g. division par 0) ou des co-processeurs associés et interruption ce qui est généré hors du processeur (par exemple un dépassement de capacité sur la pile). Dans la suite on ne fera pas la distinction.

Dans le processeur Mips, une partie spécifique du matériel sert à enregistrer les informations utiles au traitement des interruptions par le logiciel. Cette partie est référencée comme le *co-processeur 0*, pour une architecture générique on parle de *contrôleur d'interruptions*. Le co-processeur 0 contient un certain nombre de registres, nous ne décrivons ici qu'une partie de ces registres (la partie qui est implémentée dans le simulateur Spim). Voici les quatre registres décrits du co-processeur 0 :

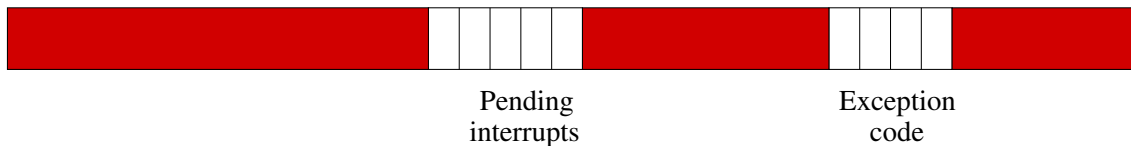
Nom du registre	Numéro du registre	Utilisation
BadVAddr	8	Registre contenant l'adresse mémoire incorrecte référencée
Status	12	Masque d'interruption et autorisation d'interruption
Cause	13	Type d'exception et bits d'interruption en attente
EPC	14	Adresse de l'instruction ayant causé l'exception

Ces quatre registres sont accédés par des instructions spéciales : `lwc0`, `mfc0`, `mtc0`, et `swc0`. Après une exception, le registre EPC contient l'adresse de l'instruction qui s'exécutait lorsque l'interruption est intervenue. Si l'instruction effectuait un accès mémoire qui a causé l'interruption, le registre BadVAddr contient l'adresse de la case mémoire référencée. Le registre Status à la configuration suivante (dans l'implémentation de Spim) :



Le masque d'interruption contient un bit pour chacun des 5 niveaux d'interruption matérielle et des 3 niveaux d'interruption logicielle. Un bit à 1 autorise une interruption à ce niveau, un bit à 0 interdit une interruption à ce niveau. Les 6 bits de poids faibles sont utilisés pour empiler (sur une profondeur de 3 au maximum) les bits `kernel/user` et `interrupt enable`. Le bit `kernel/user` est à 0 si le programme était en mode superviseur (ou mode noyau, mode privilégié) lorsque l'exception est arrivée et à 1 si le programme était en mode utilisateur. Si le bit `interrupt enable` est à 1, les interruption sont autorisées, si il est à 0 elles sont empêchées. Lorsqu'une interruption arrive, ces 6 bits sont décalés vers la gauche de 2 bits de manière à ce que les bits *courants* deviennent les bits *précédents* (les bits *vieux* sont alors perdus).

Le registre `Cause` à la structure suivante :



Les 5 bits `pending interrupt` (interruption en attente) correspondent aux 5 niveaux d'interruption. Un bit passe à 1 lorsqu'une interruption est arrivée mais n'a pas encore été traitée. Les bits `exception code` (code d'exception) décrivent la cause de l'exception avec les codes suivants :

Nombre	nom	description
0	INT	interruption externe
4	ADDRL	exception d'erreur d'adressage (load ou chargement d'instruction)
5	ADDRS	exception d'erreur d'adressage (store)
6	IBUS	erreur de bus sur chargement d'instruction
7	DBUS	erreur de bus sur load ou store
8	SYSCALL	exception d'appel système
9	BKPT	exception de breakpoint
10	RI	exception d'instruction réservée
12	OVF	dépassement de capacité arithmétique

Une exception ou une interruption provoque un saut vers une procédure qui s'appelle un gestionnaire d'interruption (*interrupt handler*). Sur le Mips, cette procédure commence systématiquement à l'adresse `80000080hex` (adresse dans l'espace d'adressage du noyau, pas dans l'espace utilisateur). Ce code examine la cause de l'exception et appelle le traitement approprié que le système d'exploitation doit exécuter. Le système d'exploitation répond à une interruption soit en terminant le processus qui a causé l'exception soit en exécutant une autre action. Un processus qui essaye d'exécuter une instruction non implémentée est tué, en revanche une exception telle qu'un défaut de page est une requête d'un processus au système d'exploitation pour réclamer le service : ramener une page du disque en mémoire. Les autres interruptions sont celles générées par les composants externes. Le travail consiste alors à envoyer ou recevoir des données de ce composant et à reprendre le processus interrompu.

Voici un exemple de code assembleur traitant une interruption sur le Mips. Le gestionnaire d'interruption sauvegarde les registres `$a0` et `$a1`, qu'il utilisera par la suite pour passer des arguments. On ne peut pas sauvegarder ces valeurs sur la pile car l'interruption peut avoir été

provoquée par le fait qu'une mauvaise valeur a été stockée dans le pointeur de pile. Le gestionnaire sauve donc ces valeurs dans deux emplacements mémoire particuliers (*save0* et *save1*). Si la procédure de traitement de l'interruption peut elle même être interrompue, deux emplacements ne sont pas suffisants. Ici, elle ne peut pas être interrompue.

```
.ktext 0x80000080
sw $a0, save0 # Handler is not re-entrant and cannot use
sw $a1, save1 # stack to save $a0, $a1
```

Le gestionnaire d'interruption charge alors les registres *Cause* et *EPC* dans les registres du CPU (ce sont des registres du co-processeur 0). Le gestionnaire n'a pas besoin de sauver *\$k0* et *\$k1* car le programme utilisateur n'est pas censé utiliser ces registres.

```
mfc0 $k0, $13      # Move Cause into $k0
mfc0 $k1, $14      # Move EPC into $k1

sgt $v0, $k0, 0x44 #ignore interrupt
bgtz $v0, done

mov $a0, $k0       # Move Cause into $a0
mov $a1, $k1       # Move EPC into $a1
jal print_excpc    #print exception message
```

Avant de revenir à la procédure originale, le gestionnaire restaure les registres *\$a0* et *\$a1*, puis il exécute l'instruction *rfe* (return from exception) qui restaure le masque d'interruption et les bits *kernel/user* dans le registre *Status*. Le contrôleur saute alors à l'instruction suivant immédiatement l'instruction qui a provoqué l'interruption.

```
done:
    lw  $a0, save0
    lw  $a1, save1
    addiu $k1, $k1, 4
    rfe
    jr  $k1

.kdata
save0: .word 0
save1: .word 0
```

## C.7 Input/Output

Il y a deux manières d'accéder à des organes d'entrée/sortie (souris, écrans, etc.) : instructions spécialisées ou périphériques placés en mémoire (*memory mapped*). Un périphérique est placé en mémoire lorsqu'il est vu comme un emplacement spécial de la mémoire, le système oriente les accès à ces adresses vers le périphérique. Le périphérique interprète ces accès comme des requêtes. On présente ici un terminal (de type VT100) utilisé par le Mips et implémenté dans le simulateur Spim. On ne détaille pas le mécanisme du clavier, uniquement celui du terminal.

Le terminal consiste en deux unités indépendantes : le transmetteur et le receveur. Le receveur lit les caractères tapés au clavier, le transmetteur écrit les caractères sur le terminal. Les caractères tapés au clavier ne sont donc affichés sur le terminal que si le transmetteur demande explicitement de le faire.

Un programme contrôle le terminal avec quatre registres mappé en mémoire : Le *registre de contrôle du receveur* est à l'adresse *fff0000<sub>hex</sub>*. Seuls deux bits sont utilisés dans ce registre, le bit 0 est appelé *ready* : si il est à 1 cela veut dire qu'un caractère est arrivé depuis le clavier mais qu'il n'a pas été encore lu par le *registre de donnée du receveur*. Ce bit ne peut pas être écrit par le programme (les écritures sont ignorées). Le bit passe de 0 à 1 lorsqu'un caractère est tapé au clavier et passe de 1 à 0 lorsque le caractère est lu par le registre de donnée du receveur.

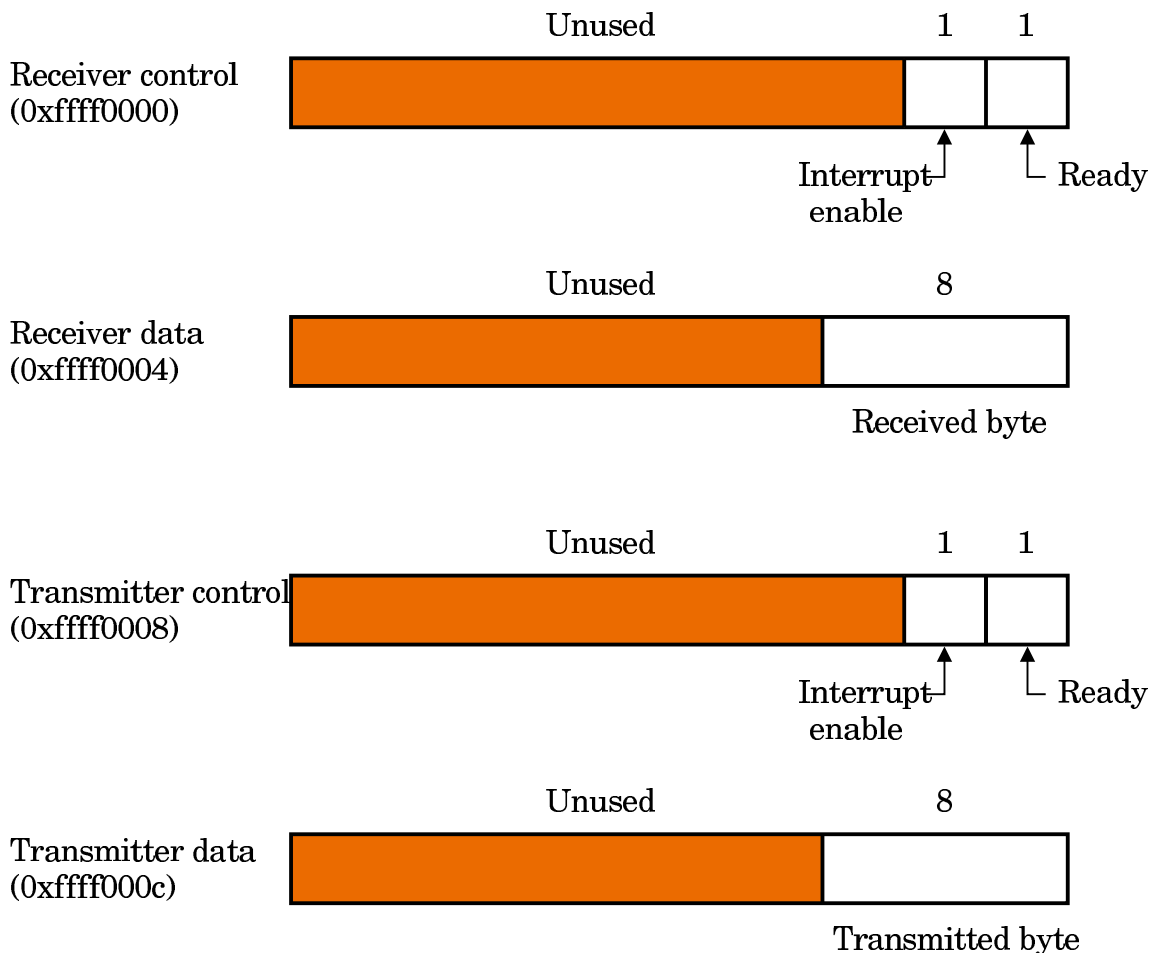
Le bit 1 du registre de contrôle du receveur est l'autorisation d'interruption du clavier (*interrupt enable*). ce bit peut être écrit ou lu par le programme. Lorsqu'il passe à 1, le terminal lève une interruption de niveau 0 dès que le bit *ready* est à 1. Pour que l'interruption soit vue par le processeur, ce niveau d'interruption doit aussi être autorisé dans le registre *Status* du co-processeur 0.

Le deuxième registre du composant est le *registre de donnée du receveur*, il est à l'adresse  $ffff0004_{hex}$ . Les 8 bits de poids faible de ce registre contiennent le dernier caractère tapé au clavier. Ce registre ne peut qu'être lu par le programme et change dès qu'un nouveau caractère est tapé au clavier. Lorsque ce registre est lu par le programme, le bit *ready* du registre de contrôle du receveur est remis à 0.

Le troisième registre du terminal est le *registre de contrôle du transmetteur* (adresse  $ffff0008_{hex}$ ). Seuls les deux bits de poids faible de ce registre sont utilisés. Il se comportent comme les bits du registre de contrôle du receveur. Le bit 0 est appelé *ready* et est en lecture seule. Si il est à 1, le transmetteur est prêt à accepter un nouveau caractère sur l'écran. Si il est à 0, le transmetteur est occupé à écrire le caractère précédent. Le bit 1 est l'autorisation d'interruption, lorsqu'il est à 1 le terminal lève une interruption dès que le bit *ready* est à 1.

Le dernier registre est le *registre de donnée du transmetteur* (adresse  $ffff000c_{hex}$ ). Quand une valeur est écrite, les 8 bits de poids faible sont envoyés à l'écran. Lorsque le registre de donnée du transmetteur est écrit, le bit *ready* du registre de contrôle du transmetteur est remis à 0. Le bit reste à 0 assez longtemps pour afficher le caractère à l'écran, puis le bit *ready* redevient 1. Le registre de donnée du transmetteur ne peut être écrit que lorsque le bit *ready* du registre de contrôle du transmetteur est à 1. Si ce n'est pas le cas l'écriture est ignorée.

Un vrai ordinateur demande un certain temps pour envoyer un caractère sur une ligne série qui connecte le terminal au processeur. Le simulateur Spim simule ce temps d'exécution lors de l'affichage d'un caractère sur la console.



## C.8 Quelques commandes utiles

**Préprocesseur** `gcc -E ex1.c -o temp.c`

Cette commande active le préprocesseur qui effectue nombre de taches simples à partir du fichier source. On a ainsi une idée de la complexité du processus généré lorsque l'on écrit trois ligne de C. La commande GNU utilisée ici par `gcc` est `cpp`.

**Code assembleur** `gcc -S ex1.c -o ex1.s`

Après cette commande, `ex1.s` contient le code assembleur sous forme lisible (comme sur la figure 3). L'assembleur généré est bien sûr celui du processeur de la machine sur lequel la commande est exécutée.

**Compilation+assembleur** `gcc -c ex1.c -o ex1.o`

Produit directement le code objet à partir du code source. L'assembleur utilisé ici par `gcc` est `as` (on peut donc exécuter de manière équivalente : `gcc -S ex1.c -o ex1.s` suivit de `as ex1.s -o ex1.o`)

**Édition de lien** `gcc ex1.o -o ex1`

Cette commade effectue l'édition de lien. La commande GNU utilisée est `ld`. Elle est difficile à utiliser directement car il faut configurer diverses options, c'est pourquoi on l'utilise à travers `gcc`.

**Compilation+assembleur+liens** `gcc ex1.c -o ex1`

Compile, assemble et effectue l'édition de lien pour produire un programme exécutable.

**Symboles utilisés** `nm ex1.o`

Liste les symboles utilisés dans le programme objet `ex1.o`. Cela est utile pour vérifier les fonctions utilisées ou définies dans une librairie. `nm` fonctionne aussi sur les codes exécutables.

**Debugage** `gcc -g ex1.c -o ex1`

Génère un exécutable contenant les informations de debugage permettant d'utiliser les debugger (`gdb`, `ddd` etc.) et de suivre l'exécution sur le code source.

**Objdump** `Objdump` permet de récupérer toutes les informations nécessaires sur les fichiers objets ou binaires. Voici les options de cette commande. Les figures 5 et 6 montrent le résultat de l'option `objdump -S`.

```
Usage: objdump <option(s)> <file(s)>
Display information from object <file(s)>.
At least one of the following switches must be given:
-a, --archive-headers    Display archive header information
-f, --file-headers      Display the contents of the overall file header
-p, --private-headers   Display object format specific file header contents
-h, --[section-]headers Display the contents of the section headers
-x, --all-headers       Display the contents of all headers
-d, --disassemble       Display assembler contents of executable sections
-D, --disassemble-all  Display assembler contents of all sections
-S, --source            Intermix source code with disassembly
-s, --full-contents     Display the full contents of all sections requested
-g, --debugging         Display debug information in object file
-e, --debugging-tags    Display debug information using ctags style
-G, --stabs              Display (in raw form) any STABS info in the file
-t, --syms              Display the contents of the symbol table(s)
-T, --dynamic-syms     Display the contents of the dynamic symbol table
-r, --reloc              Display the relocation entries in the file
```

xspim	
	PC = 00000000 EPC = 00000000 Cause = 00000000 BadVaddr = 00000000 Status = 00000000 HI = 00000000 LO = 00000000
	<b>General registers</b>
Register display	R0 (r0) = 00000000 R8 (t0) = 00000000 R16 (s0) = 00000000 R24 (t8) = 00000000 R1 (at) = 00000000 R9 (t1) = 00000000 R17 (s1) = 00000000 R25 (s9) = 00000000 R2 (v0) = 00000000 R10 (t2) = 00000000 R18 (s2) = 00000000 R26 (k0) = 00000000 R3 (v1) = 00000000 R11 (t3) = 00000000 R19 (s3) = 00000000 R27 (k1) = 00000000 R4 (a0) = 00000000 R12 (t4) = 00000000 R20 (s4) = 00000000 R28 (gp) = 00000000 R5 (a1) = 00000000 R13 (t5) = 00000000 R21 (s5) = 00000000 R29 (sp) = 00000000 R6 (a2) = 00000000 R14 (t6) = 00000000 R22 (s6) = 00000000 R30 (s8) = 00000000 R7 (a3) = 00000000 R15 (t7) = 00000000 R23 (s7) = 00000000 R31 (ra) = 00000000
	<b>Double floating-point registers</b>
	FP0 = 0.000000 FP8 = 0.000000 FP16 = 0.000000 FP24 = 0.000000 FP2 = 0.000000 FP10 = 0.000000 FP18 = 0.000000 FP26 = 0.000000 FP4 = 0.000000 FP12 = 0.000000 FP20 = 0.000000 FP28 = 0.000000 FP6 = 0.000000 FP14 = 0.000000 FP22 = 0.000000 FP30 = 0.000000
	<b>Single floating-point registers</b>
Control buttons	<div style="display: flex; flex-wrap: wrap; justify-content: space-around;"> <div style="border: 1px solid gray; padding: 2px 10px; margin: 2px;">quit</div> <div style="border: 1px solid gray; padding: 2px 10px; margin: 2px;">load</div> <div style="border: 1px solid gray; padding: 2px 10px; margin: 2px;">run</div> <div style="border: 1px solid gray; padding: 2px 10px; margin: 2px;">step</div> <div style="border: 1px solid gray; padding: 2px 10px; margin: 2px;">clear</div> <div style="border: 1px solid gray; padding: 2px 10px; margin: 2px;">set value</div> <div style="border: 1px solid gray; padding: 2px 10px; margin: 2px;">print</div> <div style="border: 1px solid gray; padding: 2px 10px; margin: 2px;">breakpt</div> <div style="border: 1px solid gray; padding: 2px 10px; margin: 2px;">help</div> <div style="border: 1px solid gray; padding: 2px 10px; margin: 2px;">terminal</div> <div style="border: 1px solid gray; padding: 2px 10px; margin: 2px;">mode</div> </div>
	<b>Text segments</b>
Text segments	<pre> [0x00400000] 0x8fa40000 lw \$4, 0(\$29) ; 89: lw \$a0, 0(\$sp) [0x00400004] 0x27a50004 addiu \$5, \$29, 4 ; 90: addiu \$a1, \$sp, 4 [0x00400008] 0x24a60004 addiu \$6, \$5, 4 ; 91: addiu \$a2, \$a1, 4 [0x0040000c] 0x00041080 sll \$2, \$4, 2 ; 92: sll \$v0, \$a0, 2 [0x00400010] 0x00c23021 addu \$6, \$6, \$2 ; 93: addu \$a2, \$a2, \$v0 [0x00400014] 0x0c000000 jal 0x00000000 [main] ; 94: jal main [0x00400018] 0x3402000a ori \$2, \$0, 10 ; 95: li \$v0 10 [0x0040001c] 0x0000000c syscall ; 96: syscall           </pre>
	<b>Data segments</b>
Data and stack segments	<pre> [0x10000000] ... [0x10010000] 0x00000000 [0x10010004] 0x74706563 0x206e6f69 0x636f2000 [0x10010010] 0x72727563 0x61206465 0x6920646e 0x726f6e67 [0x10010020] 0x000a6465 0x495b2020 0x7265746e 0x74707572 [0x10010030] 0x0000205d 0x20200000 0x616e555b 0x6e67696c [0x10010040] 0x61206465 0x65726464 0x69207373 0x6e69206e [0x10010050] 0x642f7473 0x20617461 0x63746566 0x00205d68 [0x10010060] 0x555b2020 0x696c616e 0x64656e67 0x64646120 [0x10010070] 0x73736572 0x206e6920 0x726f7473 0x00205d65           </pre>
SPIM messages	SPIM Version 5.9 of January 17, 1997 Copyright (c) 1990-1997 by James R. Larus (larus@cs.wisc.edu) All Rights Reserved. See the file README for a full copyright notice.

FIG. 4 – Interface X du simulation Spim : simulateur du jeu d'instruction du Mips



-R, --dynamic-reloc	Display the dynamic relocation entries in the file
-v, --version	Display this program's version number
-i, --info	List object formats and architectures supported
-H, --help	Display this information

**man** man 3 printf

Explique la fonction C `printf` et la bibliothèque où elle est définie. Indiquer la section de `man` (section 3) est nécessaire sinon on risque de tomber sur une commande d'un shell.

**compilation recyclable** Le compilateur `gcc` (ainsi que toutes les commandes ci-dessus) peut être recyclé pour un très grand nombre d'architectures. Pour cela, il doit être recompilé en indiquant l'architecture cible (*target*, voir le fichier `config.sub` dans les sources de la distribution de `gcc` pour la liste des architectures possibles).

ex1.o: file format pe-i386

Disassembly of section .text:

```
00000000 <.text>:
 0: 54          push    %esp
 1: 68 65 20 73 75  push    $0x75732065
 6: 6d          insl   (%dx),%es:(%edi)
 7: 20 66 72    and    %ah,0x72(%esi)
 a: 6f          outsl  %ds:(%esi),(%dx)
 b: 6d          insl   (%dx),%es:(%edi)
 c: 20 30      and    %dh,(%eax)
 e: 20 2e      and    %ch,(%esi)
10: 2e 20 20    and    %ah,%cs:(%eax)
13: 20 20      and    %ah,(%eax)
15: 31 30      xor    %esi,(%eax)
17: 30 20      xor    %ah,(%eax)
19: 69 73 20 25 64 0a 00  imul   $0xa6425,0x20(%ebx),%esi

00000020 <_main>:
20: 55          push   %ebp
21: 89 e5      mov    %esp,%ebp
23: 83 ec 18   sub    $0x18,%esp
26: 83 e4 f0   and    $0xffffffff0,%esp
29: b8 00 00 00 00 00  mov    $0x0,%eax
2e: 89 45 f4   mov    %eax,0xffffffff4(%ebp)
31: 8b 45 f4   mov    0xffffffff4(%ebp),%eax
34: e8 00 00 00 00 00  call   39 <_main+0x19>
39: e8 00 00 00 00 00  call   3e <_main+0x1e>
3e: c7 45 f8 00 00 00 00  movl   $0x0,0xffffffff8(%ebp)
45: c7 45 fc 00 00 00 00  movl   $0x0,0xffffffffc(%ebp)
4c: 83 7d fc 64  cmpl   $0x64,0xffffffffc(%ebp)
50: 7e 02      jle    54 <_main+0x34>
52: eb 15      jmp    69 <_main+0x49>
54: 8b 45 fc   mov    0xffffffffc(%ebp),%eax
57: 89 c2      mov    %eax,%edx
59: 0f af 55 fc  imul   0xffffffffc(%ebp),%edx
5d: 8d 45 f8   lea   0xffffffff8(%ebp),%eax
60: 01 10      add   %edx,(%eax)
62: 8d 45 fc   lea   0xffffffffc(%ebp),%eax
65: ff 00     incl  (%eax)
67: eb e3     jmp    4c <_main+0x2c>
69: 8b 45 f8   mov    0xffffffff8(%ebp),%eax
6c: 89 44 24 04  mov    %eax,0x4(%esp)
70: c7 04 24 00 00 00 00  movl   $0x0,(%esp)
77: e8 00 00 00 00 00  call   7c <_main+0x5c>
7c: c9        leave
7d: c3        ret
7e: 90        nop
7f: 90        nop
```

FIG. 5 – Résultat de la commande `gcc ex1.c -o ex1.o ; objdump -S ex1.o` exécutée sur un Pentium (ex1.c contient le programme de la figure 3 page 119). Le pentium est une architecture Cisc : toutes les instructions n'ont pas la même taille.

ex1MIPS.o: file format elf32-littlemips

Disassembly of section .text:

```
00000000 <main>:
 0: 27bdffe0 addiu sp,sp,-32
 4: afbf001c sw ra,28(sp)
 8: afbe0018 sw s8,24(sp)
c: 03a0f021 move s8,sp
10: afc40020 sw a0,32(s8)
14: afc50024 sw a1,36(s8)
18: afc00014 sw zero,20(s8)
1c: afc00010 sw zero,16(s8)
20: 8fc20010 lw v0,16(s8)
24: 00000000 nop
28: 28420065 slti v0,v0,101
2c: 14400003 bnez v0,3c <main+0x3c>
30: 00000000 nop
34: 0800001d j 74 <main+0x74>
38: 00000000 nop
3c: 8fc30010 lw v1,16(s8)
40: 8fc20010 lw v0,16(s8)
44: 00000000 nop
48: 00620018 mult v1,v0
4c: 00001812 mflo v1
50: 8fc20014 lw v0,20(s8)
54: 00000000 nop
58: 00431021 addu v0,v0,v1
5c: afc20014 sw v0,20(s8)
60: 8fc20010 lw v0,16(s8)
64: 00000000 nop
68: 24420001 addiu v0,v0,1
6c: 08000008 j 20 <main+0x20>
70: afc20010 sw v0,16(s8)
74: 3c040000 lui a0,0x0
78: 24840000 addiu a0,a0,0
7c: 8fc50014 lw a1,20(s8)
80: 0c000000 jal 0 <main>
84: 00000000 nop
88: 03c0e821 move sp,s8
8c: 8fbf001c lw ra,28(sp)
90: 8fbe0018 lw s8,24(sp)
94: 03e00008 jr ra
98: 27bd0020 addiu sp,sp,32
```

FIG. 6 – Résultat de la commande `gcc ex1.c -o ex1.o; objdump -S ex1.o` reciblé pour un MIPS (ex1.c contient le programme de la figure 3 page 119).