Cours de Compilation, Master 1, 2004-2005

Tanguy Risset

2 mars 2005

Table des matières

1 Généralités

Page web du cours :
`http://perso.ens-lyon.fr/tanguy.risset/cours/compil2004.html`

1.1 Biblio

La construction d'un compilateur rassemble une grande diversité de processus. Certains étant 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 le fait de maîtriser les techniques de construction d'un compilateur sont 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 assimilés 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. Parmi les ouvrages importants qui peuvent compléter ce cours je recommande . 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).

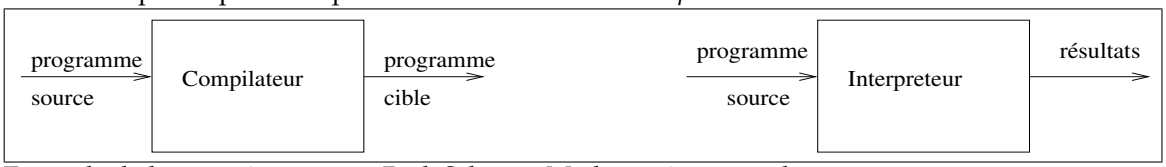
[HP98] J. Hennessy, D. Patterson *Computer Organization and Design : The hardware software interface* Morgan Kaufman 1998. Référence pour la description des architectures Risc.

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 postcript à partir d'un format donnée (ascii, latex, jpg, etc.) est aussi un compilateur. En revanche, un programme qui transforme du postcript en des pixel sur un écran est un *interpréteur*.



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

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, debboging
– 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 jeux 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).

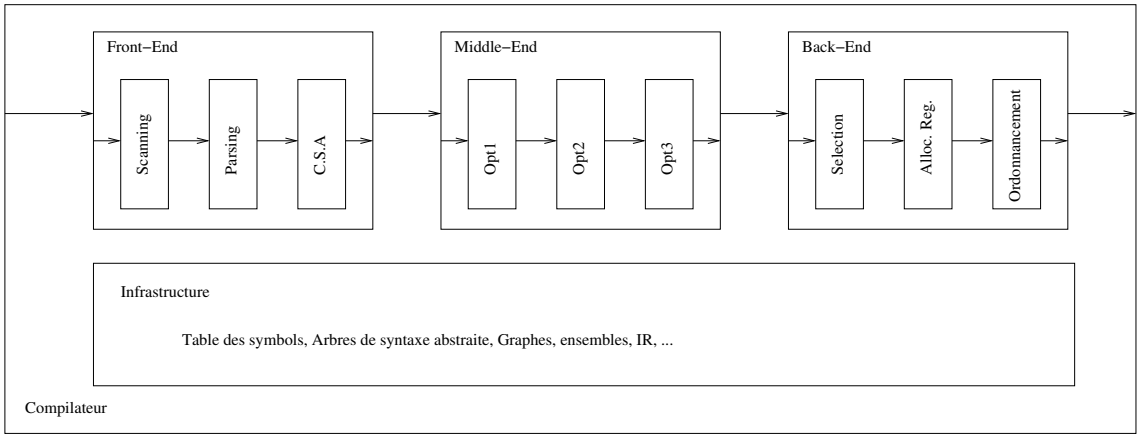
Le processus de compilation peut aussi être décomposé : comme le bytecode java qui est un programme *semi-compilé*. L'éditeur de liens, reliant le programme compilé avec les librairies précompilées qu'il utilise peut éventuellement faire des optimisations. Le programme peut même être amélioré à l'exécution 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 :

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

On décompose aujourd'hui 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 représentions. Une représentation intuitive est la suivante :



On appelle cela un compilateur optimisant (optimizing compiler) par opposition aux premier 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 francais : *Le lion mange de la viande* est syntaxiquement correcte et *le lion viande* n'est pas syntaxiquement correcte. En pratique, l'analyse 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 Francais : *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 regroupé 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 analyzeurs lexicauxs 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 :

$$^{0}\boxed{w}\boxed{x}\boxed{y}\boxed{z} \quad ,$$

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

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

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 par 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	$\Rightarrow r_w$	// load w
loadI	2	$\Rightarrow r_2$	// la constante 2 dans r_2
loadAI	r_{arp}, @x	$\Rightarrow r_x$	// load x
loadAI	r_{arp}, @y	$\Rightarrow r_y$	// load y
loadAI	r_{arp}, @z	$\Rightarrow r_z$	// load z
mult	r_w, r_2	$\Rightarrow r_w$	// $r_w \leftarrow w \times 2$
mult	r_w, r_x	$\Rightarrow r_w$	// $r_w \leftarrow (w \times 2) \times x$
mult	r_w, r_y	$\Rightarrow r_w$	// $r_w \leftarrow (w \times 2) \times x \times y$
mult	r_w, r_z	$\Rightarrow r_w$	// $r_w \leftarrow (w \times 2) \times x \times y \times z$
storeAI	r_w	$\Rightarrow r_{arp}$, @w	// écriture de w

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

loadAI	r_{arp}, @w	$\Rightarrow r_1$	// load w
add	r_1, r_1	$\Rightarrow r_1$	// $r_1 \leftarrow w \times 2$
loadAI	r_{arp}, @x	$\Rightarrow r_2$	// load x
mult	r_1, r_2	$\Rightarrow r_1$	// $r_1 \leftarrow (w \times 2) \times x$
loadAI	r_{arp}, @y	$\Rightarrow r_2$	// load y
mult	r_1, r_2	$\Rightarrow r_1$	// $r_w \leftarrow (w \times 2) \times x \times y$
loadAI	r_{arp}, @z	$\Rightarrow r_2$	// load z
mult	r_1, r_2	$\Rightarrow r_1$	// $r_1 \leftarrow (w \times 2) \times x \times y \times z$
storeAI	r_1	$\Rightarrow r_{arp}$, @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 intermédiare utilisée dans le MSCP (Massively Scalar Compiler Project),développé à l'université de Rice. Iloc est détaillé en Annexe A (page 117). 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 $r_1 \Rightarrow r_2$: charger dans r_2 le contenu de la case dont l'adresse est contenue dans r_1,
- chargement.
  loadI $2 \Rightarrow r_2$: charger dans r_2 la valeur 2.
- adressage indirect *Address-Immediate*.
  loadAI r_1, $2 \Rightarrow r_2$: charger dans r_2 le contenu de la case dont l'adresse est "contenu de r_1+2".
- adressage indirect indexé *Adress-Offset*.
  loadAO r_1, $r_2 \Rightarrow r_3$: charger dans r_3 le contenu de la case dont l'adresse est "contenu de r_1+contenu de r_2".[1]

Chaque instruction ILOC peut être précédée d'un label qui peut être référencé par les instructions de branchement (jump,cbr). Le registre r_0 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.

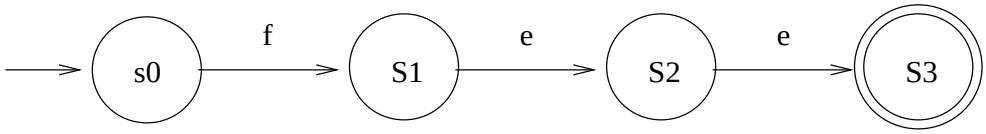
[1] demander le nom des différents adressage en francais

2 Grammaires et expressions régulières

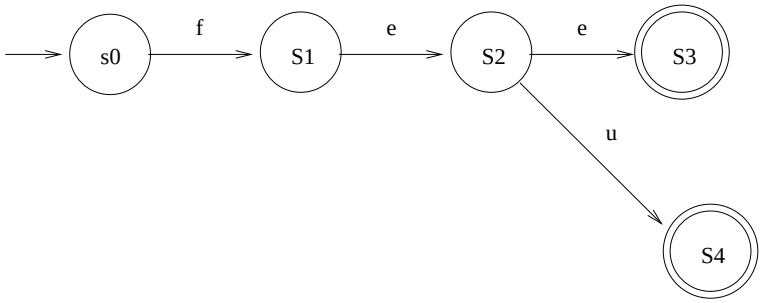
2.1 Quelques définitions

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 reconnut 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 ;*
- Σ *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*

□

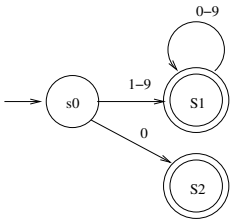
Dans notre exemple de l'automate reconnaissant ”fee”, on a $S = \{s_0, s_1, s_2, s_3\}$, $\Sigma = \{f, e\}$ (ou tout l'alphabet) $\delta = \{\delta(s_0, f) \rightarrow s_1, \delta(s_1, e) \rightarrow s_2, \delta(s_2, e) \rightarrow s_3\}$. Il y a en fait un état implicite d'erreur (s_e) vers lequel 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 $x_1 x_2 \dots x_n$, 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

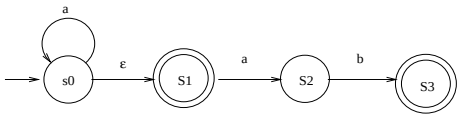
reconnaît n'importe quel entier :



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 deux manières de représenter cela sur les diagrammes de transition : soit on autorise les ϵ-transitions, c'est à dire des transitions du type : $\delta(s_i, \epsilon) = s_j$ où ϵ 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 δ 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 $x_0 x_1 \ldots x_n$ 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 ϵ 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 déterminiser 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 Σ (éventuellement augmenté de la chaîne vide ϵ). 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 $\{s \mid s \in R$ ou $s \in S\}$.
- la concaténation de deux ensembles R et S notée RS est $\{rs \mid r \in R$ et $s \in S\}$. On note R^i pour $RRR \ldots R$ 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 ϵ) 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 précédence (priorité de précédence : fermeture $>$ concaténation $>$ union). Enfin, on utilise des raccourcis de notation : par exemple $[0..4]$ équivaut à $0|1|2|3|4$, 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 une lettre suivi de lettres ou de chiffres. On peut décrire cet ensemble de chaînes par l'expression régulière : $[a..z]([a..z]|[0..9])^*$.

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

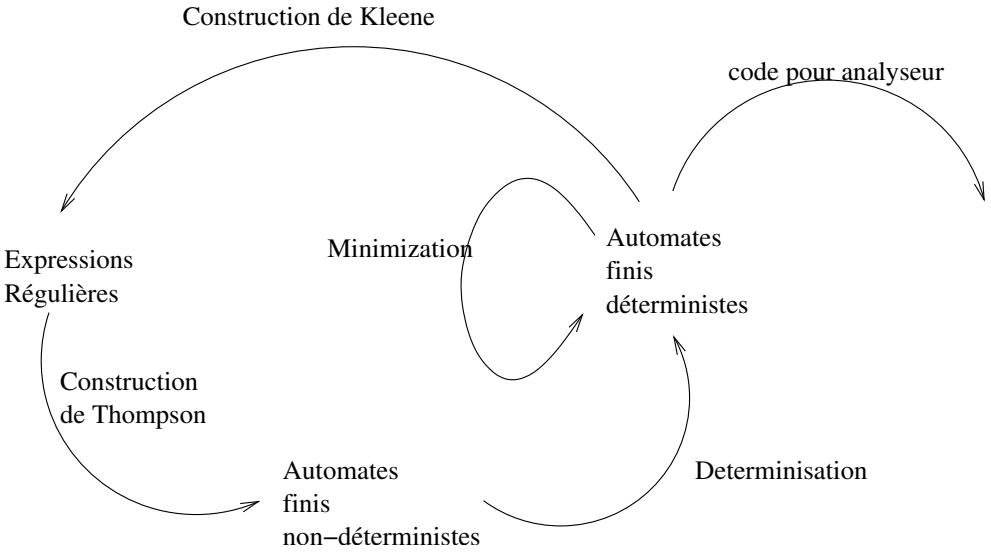
Pour les réels $(+ \mid - \mid \epsilon)(0 \mid [1..9][0..9]^*)(.[0..9]^* \mid \epsilon)E(+ \mid - \mid \epsilon)(0 \mid [1..9][0..9]^*)$

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

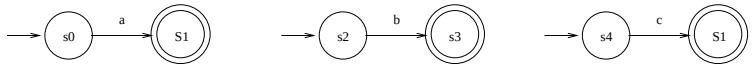
La théorie des langages réguliers permet de générer automatique un analyseur syntaxique à partir d'une expression régulière. Le schéma global est le suivant :

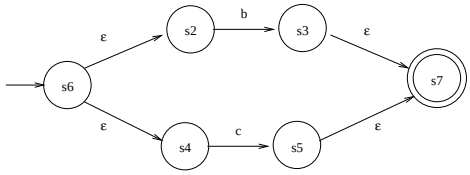


2.2.1 Des expressions régulières aux automates non déterministes

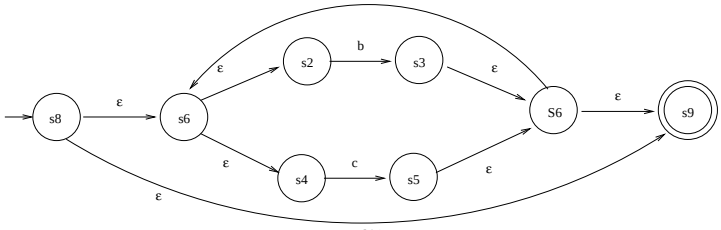
Pour être capable de construire un automate qui reconnaisse un langage régulier quelconque, il suffit d'avoir un mécanisme qui reconnaît chaque lettre, puis un mécanisme qui reconnaît la concaténation, l'union et la fermeture de langages reconnus.

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

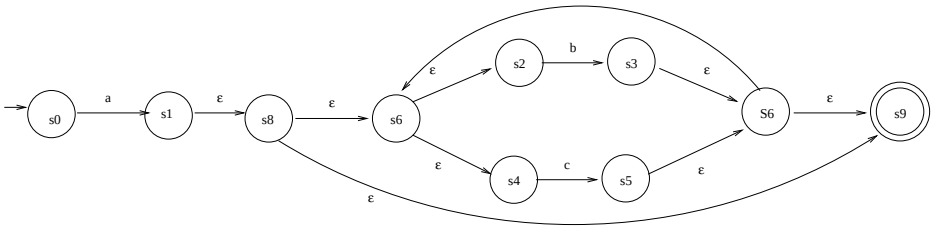




NFA pour b|c

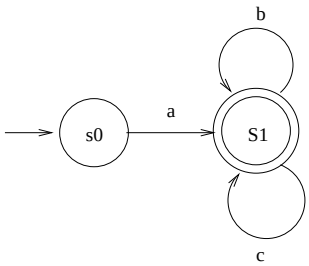


NFA pour (b|c)*



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

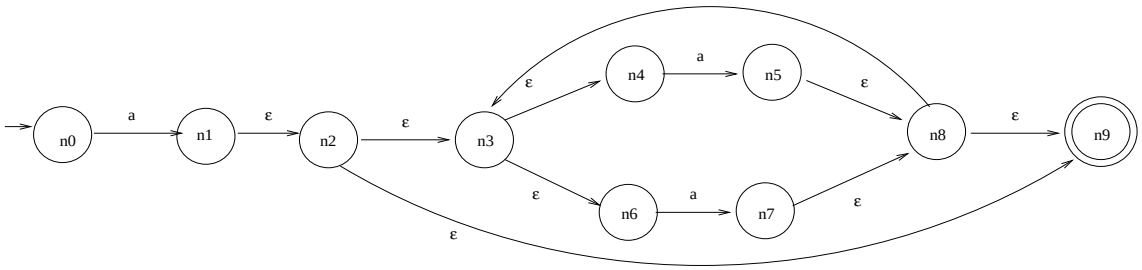
On veut maintenant obtenir un automate déterministe $(D, \Sigma, \delta_D, d_0, D_F)$ à partir d'un automate non-déterministe $(N, \Sigma, \delta_N, n_0, N_F)$. La clef est de dériver D et δ_D. Σ reste le même. L'algorithme utilisé est appelé la *construction de sous-ensembles* (subset construction). L'idée c'est que les états du nouvel automate sont des ensembles d'états de l'ancien automate. Un nouvel état $q_i = \{n_1, \ldots, n_k\}$ 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 :

$q_0 \leftarrow \epsilon$-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 ;

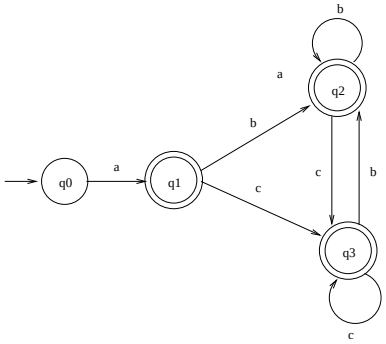
L'ϵ-fermeture d'un ensemble d'états ajoute à cet ensemble d'état tous les états pouvant être atteint par des ϵ-transitions à partir des états de l'ensemble. L'opérateur $\Delta(q_i, c)$ calcule l'ensemble des états atteignables en lisant c depuis les états n^i_j de q_i : si $q_i = \{n^i_1, \ldots, n^i_k\}$ alors $\Delta(q_i, c) = \cup_{n^i_j \in q_i} \delta_N(n^i_j, c)$

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



1. l'ϵ-fermeture de n_0 donne l'état $q_0 = \{n_0\}$.
2. la première itération de la boucle while calcule :
   - $\Delta(q_0, a) = \{n_1\}$; ϵ-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 :
   - $\Delta(q_1, a) = \emptyset$
   - $\Delta(q_1, b) = \{n_5\}$; ϵ-fermeture($\{n_5\}$)=$\{n_5, n_8, n_9, n_3, n_4, n_6\} = q_2$
   - $\Delta(q_1, c) = \{n_7\}$; ϵ-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'agorithme s'arrête en $|Q|$ itérations au maximum, donc l'algorithme présenté s'arrête bien.

On calcule l'ϵ-fermeture par un autre algorithme point fixe.

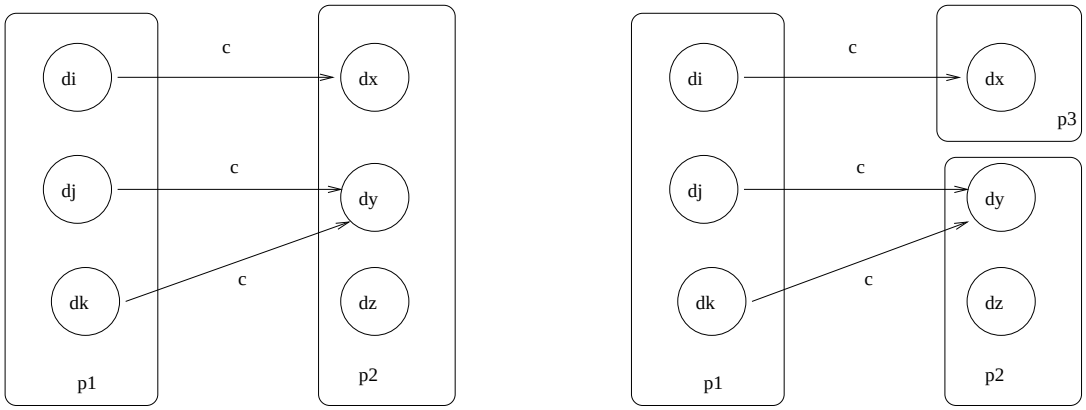
2.2.3 Minimisation

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 :

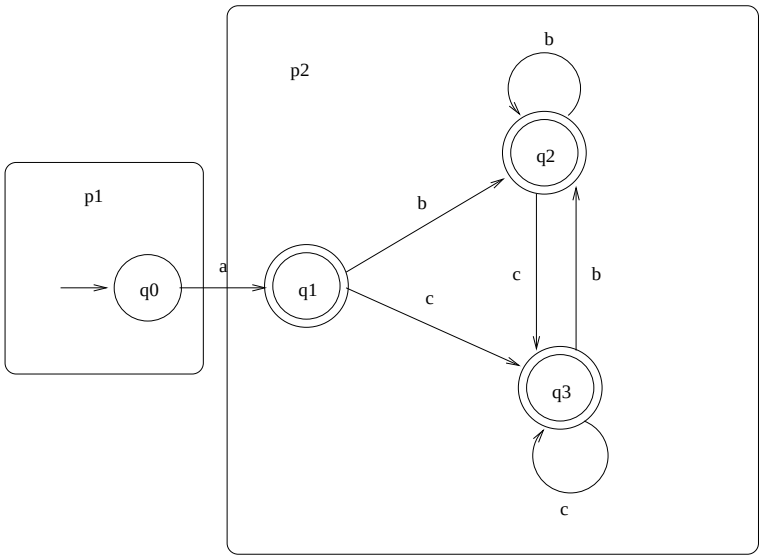
$P \leftarrow \{D_F, D - D_F\}$
tant que (P change)
    $T \leftarrow \emptyset$
    pour chaque ensemble $p \in P$
        $T \leftarrow T \cup$ Partition(p)
    $P \leftarrow T$

Partition(p)
    pour chaque $c \in \Sigma$
        si c sépare p en $\{p_1, \ldots, p_k\}$
            alors Return($\{p_1, \ldots, p_k\}$)
    Return(p)

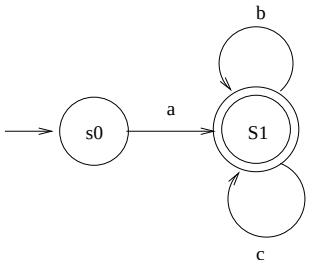
La procédure Partition découpe l'ensemble p en mettant ensemble les états p_i de p qui arrivent dans le même ensemble. Par exemple, le schéma ci-dessous présente deux cas de figure. Sur la gauche, on a un automate dans lequel le caractère `c` ne sépare pas la partition p_1 car toutes les transitions étiquetées par `c` arrive 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 :

$char \leftarrow$ prochain caractère
state $\leftarrow s_0$
tant que ($char \neq EOF$)
    $state \leftarrow \delta(state, char)$
    $char \leftarrow$ prochain caractère
si ($state \in S_F$)
    alors accepter
    sinon rejeter

δ	a	b	c	autres
s_0	s_1	–	–	–
s_1	–	s_1	s_1	–
–	–	–	–	–

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 :

s_0 : $char \leftarrow$ prochain caractère
si ($char =' a'$)
alors goto s_1
sinon goto s_e

s_1 : $char \leftarrow$ prochain caractère
si ($char =' b' | char =' c'$)
alors goto s_1
sinon si ($char =' EOF'$)
alors accepter
sinon goto s_e

s_e : rejeter

Ce type d'implémentation devient rapidement difficile à gérer manuellement, 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

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 délicat à analyser. Par exemple, en Fortran 77, les espaces ne sont pas significatifs, donc `unentier, un entier, u n e n t i e r` représentent la même chaîne. Les identificateurs sont limités à 6 carctères, 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.

3 Analyse syntaxique

Dans ce cours nous allons étudier comment produire des analyseurs syntaxique pour les langages définis par des *grammaires hors contexte*. Il existe de nombreuses méthodes pour *parser* (i.e. analyser syntaxiquement) 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. Le même principe est utilisé pour les parser $LL(1)$. Nous verrons aussi une technique ascendante (*bottom-up* $LR(1)$ *parsing*) basée sur des tables aussi appelée *shift reduce* parsing.

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 mots découvert 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 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 $\underline{elem}$ correspond à une phrase du langage décrit par cette grammaire. Par exemple, $\underline{elem}\ \ \underline{elem}$ 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 :

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

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

$$\begin{array}{lcl} Start & \rightarrow & Parent \\ & | & Croch \end{array}$$

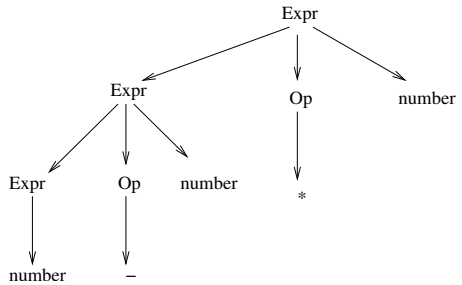
Considérons la grammaire suivante :

$$\begin{array}{|lcl} 1. & Expr & \rightarrow\ Expr\ \ Op\ \ \underline{nombre} \\ 2. & & \rightarrow\ \underline{nombre} \\ 3. & Op & \rightarrow\ \underline{+} \\ 4. & & \rightarrow\ \underline{-} \\ 5. & & \rightarrow\ \underline{\times} \\ 6. & & \rightarrow\ \underline{\div} \end{array}$$

En appliquant la séquence de règles : 1, 5, 1, 4, 2 on arrive à dériver la phrase : $nombre - nombre \times nombre$:

$Règle$	$Phrase$
	$Expr$
1	$Expr\ Op\ number$
5	$Expr\ \times\ number$
1	$Expr\ Op\ number\ \times\ number$
4	$Expr\ -\ number\ \times\ number$
2	$number\ -\ number\ \times\ number$

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

$Règle$	$Phrase$
	$Expr$
1	$Expr\ Op\ number$
1	$Expr\ Op\ number\ Op\ number$
2	$number\ Op\ number\ Op\ number$
4	$number\ -\ number\ Op\ number$
5	$number\ -\ number\ \times\ number$

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	$Instruction$	$\rightarrow$	$\underline{if}\ Expr\ \underline{then}\ Instruction\ \underline{else}\ Instruction$
2		$\mid$	$\underline{if}\ Expr\ \underline{then}\ Instruction$
3		$\mid$	$Assignation$
4		$\mid$	$AutreInstructions....$

Avec cette grammaire, le code

$$if\ Expr_1\ then\ if\ Expr_2\ then\ Ass_1\ else\ Ass_2$$

peut être compris de deux manières :

$if\ Expr_1$
  $then\ if\ Expr_2$
    $then\ Ass_1$
    $else\ Ass_2$

$if\ Expr_1$
  $then\ if\ Expr_2$
    $then\ Ass_1$
  $else\ Ass_2$

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 :

$$
\begin{array}{|rlcl}
1 & Instruction & \rightarrow & AvecElse \\
2 & & | & DernierElse \\
3 & AvecElse & \rightarrow & \underline{if}\ Expr\ \underline{then}\ AvecElse\ \underline{else}\ AvecElse \\
4 & & | & Assignation \\
5 & & | & AutreInstructions.... \\
6 & DernierElse & \rightarrow & \underline{if}\ Expr\ \underline{then}\ Instruction \\
7 & & | & \underline{if}\ Expr\ \underline{then}\ AvecElse\ \underline{else}\ DernierElse \\
8 & & | & AutreInstructions....
\end{array}
$$

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 $(nombre - nombre) \times nombre$ alors que l'on voudrait évaluer $nombre - (nombre \times nombre)$.

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

$$
\begin{array}{|rlcl}
1. & Expr & \rightarrow & Expr\ \underline{+}Term \\
2. & & \rightarrow & Expr\ \underline{-}Term \\
3. & & \rightarrow & Term \\
4. & Term & \rightarrow & Term\ \underline{\times}\ number \\
5. & & \rightarrow & Term\ \underline{\div}\ number \\
6. & & \rightarrow & number
\end{array}
$$

On fera de même pour introduire les précendences 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)

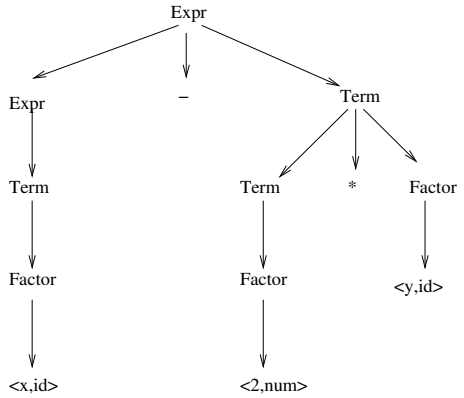
La grammaire d'expressions arithmétiques classiques :

$$
\begin{array}{|rlcl}
1. & Expr & \rightarrow & Expr\ \underline{+}\ Term \\
2. & & \rightarrow & Expr\ \underline{-}\ Term \\
3. & & \rightarrow & Term \\
4. & Term & \rightarrow & Term\ \underline{\times}\ Factor \\
5. & & \rightarrow & Term\ \underline{\div}\ Factor \\
6. & & \rightarrow & Factor \\
7. & Factor & \rightarrow & \underline{(}\ Expr\ \underline{)} \\
8. & & \rightarrow & number \\
9. & & \rightarrow & identifier
\end{array}
$$

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 :

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

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



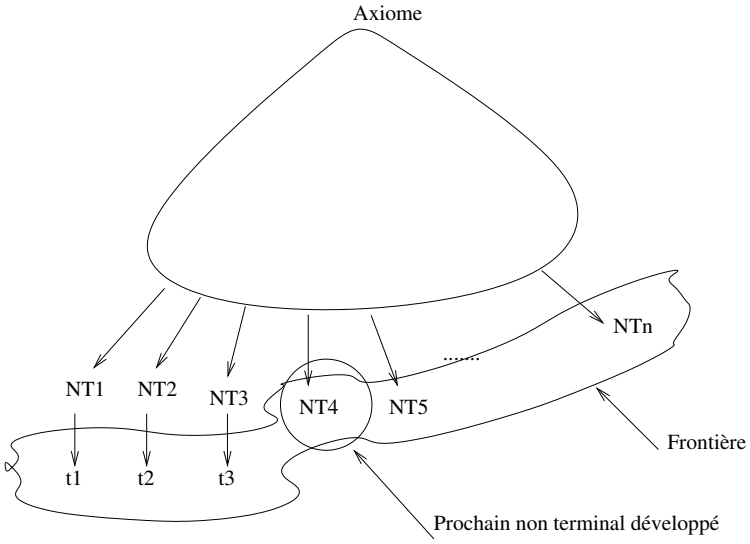
Les parseurs descendants (*top down parsers*) vont partir de l'axiome et à chaque étape choisir d'appliquer une règle pour l'un des non terminals présents sur la *frontière* de l'arbre. Les parsers ascendants (*bottom up parsers*) vont partir des feuilles et tenter de reconstruire l'arbre à partir du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

Les grammaires hors contexte décrivent plus de langages que les expressions régulières (celui des parenthèses équilibrées ne peut être exprimé en expression régulière, les prioritées entre opérateurs peuvent être exprimées dans la structure des grammaires hors contexte). La classe des grammaires *régulières* décrit exactement les même 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. As-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.).

3.2 Analyse syntaxique descendante

Le principe de l'analyse syntaxique descendante est le suivant : on commence avec l'axiome de la grammaire et on développe systématiquement l'arbre de dérivation jusqu'aux feuilles (terminaux) en choisissant 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 mots qui sont dans la phrase ; si c'est OK, on continue, sinon, il faut revenir en arrière (*bactracker*) 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 bactracker. 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 a 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 :

$Règle$	$Phrase$	$Entree$
	$Expr$	$\uparrow x - 2 \times y$
1	$Expr + Term$	$\uparrow x - 2 \times y$
1	$Expr + Term + Term$	$\uparrow x - 2 \times y$
1	$Expr + Term$ ...	$\uparrow x - 2 \times y$
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 :

$Règle$	$Phrase$	$Entree$
	$Expr$	$\uparrow x - 2 \times y$
3	$Term$	$\uparrow x - 2 \times y$
6	$Factor$	$\uparrow x - 2 \times y$
9	$identifier$	$\uparrow x - 2 \times y$
$\rightarrow$	$identifier$	$x \uparrow -2 \times y$

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. Dans le doute, on backtrack 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	$identifier \ - \ Term$	$\uparrow x - 2 \times y$
$\rightarrow$	$identifier \ - \ Term$	$x \uparrow -2 \times y$
$\rightarrow$	$identifier \ - \ Term$	$x- \uparrow 2 \times y$
4	$identifier \ - \ Term \ \times \ Factor$	$x- \uparrow 2 \times y$
6	$identifier \ - \ Factor \ \times \ Factor$	$x- \uparrow 2 \times y$
8	$identifier \ - \ number \ \times \ Factor$	$x- \uparrow 2 \times y$
$\rightarrow$	$identifier \ - \ number \ \times \ Factor$	$x - 2 \uparrow \times y$
$\rightarrow$	$identifier \ - \ number \ \times \ Factor$	$x - 2\times \uparrow y$
9	$identifier \ - \ number \ \times \ identifier$	$x - 2\times \uparrow y$
$\rightarrow$	$identifier \ - \ number \ \times \ identifier$	$x - 2 \times y \uparrow$

3.2.2 Élimination de la récursion à gauche

La récursivité à gauche pose un problème pour ce type de parser car le bactrack 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 $A \rightarrow \epsilon$.

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

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 :

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

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 :

$$\begin{array}{rlcl} 1. & Expr & \rightarrow & Term \ Expr' \\ 2. & Expr' & \rightarrow & \underline{+} \ Term \ Expr' \\ 3. & & \rightarrow & \underline{-} \ Term \ Expr' \\ 4. & & \rightarrow & \epsilon \\ 5. & Term & \rightarrow & Factor \ Term' \\ 6. & Term' & \rightarrow & \underline{\times} \ Factor \ Term' \\ 7. & & \rightarrow & \underline{\div} \ Factor \ Term' \\ 8. & & \rightarrow & \epsilon \\ 9. & Factor & \rightarrow & \underline{(} \ Expr \ \underline{)} \\ 10. & & \rightarrow & number \\ 11. & & \rightarrow & identifier \end{array}$$

3.2.3 Élimination du bactrack

On va montrer comment on peut, pour des grammaires récursives à droite, systématiquement éliminer le bactrack. En fait, le bactrack arrive lorsque l'algorithme choisi une mauvaise règle à

appliquer. Si l'algorithme choisissait systématiquement la bonne règle, il n'y aurait jamais besoin de bactracker.

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 mot 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 mot 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* .

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 mot 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 α* □

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

2.	$Expr'$	$\rightarrow$	$\underline{+}\ Term\ Expr'$
3.		$\rightarrow$	$\underline{-}\ Term\ Expr'$
4.		$\rightarrow$	ϵ

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

Si un ensemble $First(A)$ contient ϵ, on ne sait pas quel non terminal sera recontré lorsque ce A aura été développé. Dans ce cas, le parser doit connaître ce qui peut apparaître immédiatement à la droite de la chaine vide ϵ 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 : $Follow(Expr') = \{eof, \underline{)}\}$. 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 $A \rightarrow \beta_1\beta_2 \ldots \beta_k$
        $First(A) \leftarrow First(A) \cup (First(\beta_1) - \{\epsilon\})$
        $i \leftarrow 1$
        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 $A \rightarrow \beta_1\beta_2 \ldots \beta_k$
        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. Les ensemble $First$ et $Follow$ pour les non terminaux de la grammaire d'expressions arithmétiques classiques (variante récursive à droite) sont :

$symbole$	$First$	$Follow$
$Expr$	$\{\underline{(}, number, identifier\}$	$\{\underline{)}\}$
$Expr'$	$\{\underline{+}, \underline{-}, \epsilon\}$	$\{\underline{)}\}$
$Term$	$\{\underline{(}, number, identifier\}$	$\{\underline{+}, \underline{-}\}$
$Term'$	$\{\underline{\times}, \underline{\div}, \epsilon\}$	$\{\underline{+}, \underline{-}\}$
$Factor$	$\{\underline{(}, number, identifier\}$	$\{\underline{\times}, \underline{\div}, \underline{+}, \underline{-}\}$

On peut maintenant formuler la condition à laquelle une grammaire permet d'éviter les backtrack dans un parsing descendant. Notons $First^+(\alpha)$ l'ensemble $First(\alpha)$ si $\epsilon \notin First(\alpha)$ et $First(\alpha) \cup Follow(\alpha)$ si $\epsilon \in First(\alpha)$. La condition que doit vérifier la grammaire est que pour tout non-terminal A qui est en partie de gauche plusieurs règles : $A \rightarrow \beta_1 \mid \beta_2 \mid \ldots \beta_n$ on doit avoir la propriété suivante. Par extension $First^+(ABCD)$ désigne $First^+(A)$ et dans le cas particulier où un des β_i est égal à ϵ, on rajoute l'intersection avec des $First^+(\beta_i)$ avec $Follow(A)$.

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

Pour que l'on ait cette propriété, étant donnée une grammaire récursive 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 :

$$A \rightarrow \alpha \;\; \beta_1 \mid \alpha \;\; \beta_2 \mid \ldots \mid \alpha \;\; \beta_n \mid \gamma_1 \mid \ldots \mid \gamma_k$$

on les remplacera par :

$$A \rightarrow \alpha \ B \mid \gamma_1 \mid \ldots \mid \gamma_k$$
$$B \rightarrow \beta_1 \mid \ \beta_2 \mid \ldots \mid \beta_n$$

La factorisation à gauche permet de convertir certaines grammaires nécessitant du bactrack en des grammaires ne nécessitant pas de bactrack (ou grammaires prédictives, ou grammaire LL(1)). Cependant toutes les grammaires hors contexte ne peuvent pas être transformées en grammaires ne nécessitant pas de bactrack. 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 ($motCourant = a_1$) alors
$motCourant = prochainMot()$;
return ($trouver\alpha_1()$) /* $A \rightarrow a_1 \ \alpha_1$ choisi */
sinon si ($motCourant \in First^+(\beta_1)$) alors
return ($trouver\beta_1() \wedge trouver\beta_2()$) /* $A \rightarrow \beta_1 \ \beta_2$ choisi */
sinon si ($motCourant \in Follow(A)$)
alors return True /* $A \rightarrow \epsilon$ choisi */
sinon return False /* erreur */

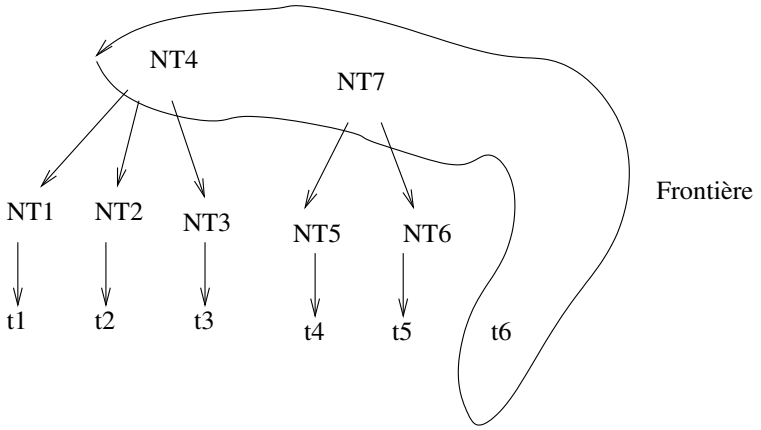
Par exemple, voici le parser récursif descendant pour la grammaire des expressions arithmétiques (variante récursive à droite) :

```
Main()
    /* But → Expr */
    motCourant = prochainMot() ;
    si (Expr() ∧ motCourant == 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 (motCourant == +)∨
            (motCourant == −) alors
        motCourant = prochainMot()
        si (Term() == False)
            alors retrun 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 (motCourant == ×)∨
        (motCourant == ÷) alors
        motCourant = prochainMot()
        si (Factor() == False)
            alors retrun False
            sinon return TPrime()
    /* Term' → ϵ */
    return True

Factor()
    /* Factor → ( Expr ) */
    si (motCourant == () alors
        motCourant = prochainMot()
        si (Expr() == False)
            alors retrun False
            sinon si (motCourant ≠))
                alors return False
    sinon
    /* Factor → number */
    /* Factor → identifier */
        si (motCourant() ≠ number)∧
            (motCourant() ≠ identifier)
        alors return False
    motCourant = prochainMot()
    return True
```

On peut aussi imaginer une implémentation à base de table indiquant, pour chaque non terminal, la règle à appliquer en fonction du mot rencontré. Ce type de parseur présentent les avantages d'être facilement codés manuellement, ils sont compacts et efficaces. Ils génèrent en particulier un debogging très efficace pour les entrées non conformes.

3.3 Parsing ascendant

On va voir maintenant l'approche inverse : on construit des feuilles au fur et à mesure qu'on rencontre des mots, 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'ou leur nom : *Left-to -right scan, Reverse-rightmost derivation with 1 symbol lookahead*. On appelle aussi cela du *shift-reduce* parsing.

On introduit la notation suivante : $\langle A \rightarrow \beta, k\rangle$ indique que sur la frontiere, à la position k, il y une possibilité de réduction avec la règle $A \rightarrow \beta$. On appelle cela une *réduction candidate* (*handle* en anglais). L'implémentation du parser va utiliser une pile. Le parser fonctionne de la manière suivante : il empile la frontière, réduit les réductions candidates à partir du sommet de pile (dépile les parties droites et empile la partie gauche de la règle selectionnée). Quand cela n'est plus possible, il demande le prochain mot et l'empile puis recommence. Le sommet de la pile correspond à la partie la plus à droite de la frontière. L'algorithme générique pour le *shift-reduce* parsing est le suivant :

push $Invalid$
$motCourant \leftarrow prochainMot()$
tant que ($motCourant \neq eof$ ou la pile ne contient pas exactement $Expr$ au dessus de $Invalid$)
    si une réduction candidate $\langle A \rightarrow \beta, k\rangle$ est au sommet de pile
    alors /* reduire par $A \rightarrow \beta$ */
        pop $|\beta|$ symbols
        push A
    sinon si ($motCourant \neq eof$)
        alors/* shift : décalage d'un sur l'entrée */
            push $motCourant$
            $motCourant \leftarrow prochainMot()$
        sinon /* pas de réduction candidate pas d'input */
            error

Ici les bonnes réductions candidates ont été trouvées par un oracle. Une détection efficace des bonnes réductions candidates à réaliser est la clef d'un parsing $LR(1)$ efficace. Une fois que l'on a cela, le reste est trivial.

3.3.1 Détection des réductions candidates

On va introduire une nouvelle notation pour les réductions candidates : $\langle A \rightarrow \beta\bullet\rangle$. Le point noir ($\bullet$, *dot* en anglais) représente le sommet de la pile. La notation $\langle A \rightarrow \beta\bullet\rangle$ signifie, la pile est telle que l'on peut réduire avec la règle $A \rightarrow \beta$ à partir du sommet de pile (on n'a plus besoin de l'adresse dans la frontière puisqu'on choisi de réduire toujours 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 α, si jamais j'empile encore β 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. Un ensemble de réductions candidates ou potentielles représente donc un état du parser (i.e. un ensemble de possibilités d'action à réaliser en fonction de ce qui est lu). Les parseurs $LR(1)$ sont en fait des automates dont les états sont constitués d'ensemble de réductions potentielles ou candidates. On a vu qu'on pouvait implémenter ces automates à partir d'une table repésentant la fonction de transition, les outils permettant la construction de cette table sont très largement utilisés aujourd'hui pour la construction de parser.

La grosse difficulté va être de construire deux tables ($Action$ et $Goto$) qui vont 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
$motCourant \leftarrow prochainMot()$
tant que (True)
    $s \leftarrow$ sommet de pile
    si $Action[s, motCourant]$= "shift"
    alors push $motCourant$
        push $Goto[s, motCourant]$
        $motCourant \leftarrow prochainMot()$
    sinon si $Action[s, motCourant]$=" reduce $A \rightarrow \beta$"
        alors pop $2 \times |\beta|$ symboles
            $s \leftarrow$ sommet de pile
            push A
            push $Goto[s, A]$
        sinon si $Action[s, motCourant]$= "accept" alors Accept
            sinon Erreur

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, et au lieu d'empiler uniquement la frontière, on empile aussi l'état courant.

3.3.2 Construction des tables $LR(1)$

Pour construire les tables $Action$ et $Goto$, le parseur construit un automate permettant de reconnaître les réductions potentielles et candidates et déduit 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)* : $\mathcal{CC} = \{cc_0, cc_1, \ldots, cc_n\}$. 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).* □

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, a est le prochain caractère 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. $\quad\ \ | \ \underline{elem}$

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

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

On suppose, comme c'est le cas ici que l'on a une unique règle indiquant le but à réaliser pour reconnaître une phrase du langage. L'item $[But \rightarrow \bullet\ List, eof]$ sert à initaliser l'état initial cc_0 du parseur. On va rajouter a cela les item qui peuvent être rencontrés pour réaliser cet 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 $C \rightarrow \gamma \in P$
        pour chaque $b \in First(\delta a)$
          $s \leftarrow s \cup \{[C \rightarrow \bullet\gamma, b]\}$

Voici l'explication en francais : 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 δa. Si cette réduction (par exemple $C \rightarrow \gamma$) a lieu, on va donc recontrer γ 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 initalement $[But \rightarrow \bullet\ List, eof]$ dans cc_0, le passage par la procédure $closure()$ ajoute deux items : $[List \rightarrow \bullet\ \underline{elem}\ List, eof]$ et $[But \rightarrow \bullet\ \underline{elem}, eof]$, comme les $\bullet$ précèdent uniquement des non terminaux, cela ne génère pas plus d'item :

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

Table *Goto* On construit une table qui prévoit l'état que le parseur devrait prendre s'il était dans un certain état $s = cc_i$ et qu'il lisait un symbole x.

$goto(s, x)$
  $moved \leftarrow \emptyset$
  pour chaque item $i \in s$
    si i est $[\alpha \rightarrow \beta \bullet x\delta, a]$ alors
      $moved \leftarrow moved \cup \{[\alpha \rightarrow \beta x \bullet \delta, a]\}$
  return closure($moved$)

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 :

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

On a aussi

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

et

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

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

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

$cc_0 \leftarrow closure(s_0)$
$\mathcal{CC} \leftarrow cc_0$
tant que ($\mathcal{CC}$ change)
    pour chaque $cc_j \in \mathcal{CC}$ non marqué
        marquer cc_j
        pour chaque x suivant un $\bullet$ dans un item de cc_j
            $temp \leftarrow goto(cc_j, x)$
            si $temp \notin \mathcal{CC}$
                alors $\mathcal{CC} \leftarrow \mathcal{CC} \cup \{temp\}$
            enregistrer la transition de cc_j à $temp$ sur x

L'ensemble $\mathcal{CC}$ 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 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 $Goto$ (β et γ peuvent être ϵ).
2. Un item de la forme $[A \rightarrow \beta\bullet, a]$ indique que le parser a reconnu β ; 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 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. On récupère donc les deux tables :

$Table\ \ Action$		
$etat$	$\underline{eof}$	$\underline{elem}$
cc_0		$shift\ \ cc_2$
cc_1	$accept$	
cc_2	$reduce\ \ List \rightarrow \underline{elem}$	$shift\ \ cc_2$
cc_3	$reduce\ \ List \rightarrow\ \ \underline{elem}\ \ List$	

$Table\ \ Goto$			
$etat$	$List$	$\underline{eof}$	$\underline{elem}$
cc_0	cc_1		cc_2
cc_1			
cc_2	cc_3		cc_2
cc_3			

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 item contient à la fois $[A \rightarrow \alpha \bullet \underline{c}\gamma, \underline{a}]$ et $[B \rightarrow\ \beta\bullet, \underline{c}]$. Il peut aussi contenir un conflit $reduce - reduce$: $[A \rightarrow \alpha\bullet, \underline{a}]$ et $[B \rightarrow \beta\bullet, \underline{a}]$. La meilleure méthode pour déterminer si une grammaire est $LR(1)$ est de lancer le 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.

Gestion d'erreur Le parseur est largement utilisé pour débugger 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 *mots de synchronisation* qui permettent de synchroniser l'entrée lue avec l'état interne du parseur. Lorsque le parseur rencontre des erreurs, il oublie les mots jusqu'à ce qu'il rencontre un mot de synchronisation. Par exemple, dans beaucoup de langage "à la Algol" le point-virgule est un séparateur qui peut être utilisé comme mot de synchronisation.

Pour les parseurs descendants, ca 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 caractères 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).

Ambiguité 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 le 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 $List \rightarrow List\ \underline{elem}$ on produit un AST qui descend à gauche et donc : $((elem_1\ elem_2)\ elem_3)\ldots$. 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

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 inchangé). 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 $\mathcal{CC}$ 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 faîte en testant explicitement la valeur du symbole à postériori.

De gros efforts sont faits 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

Enfin, 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 item sont alors de la forme $\langle A \rightarrow \alpha \bullet \gamma \rangle$ sans le symbole censé suivre après γ. L'algorithme est similaire à $LR(1)$ mais il utilise l'ensemble $Follow$ entier (indépendant du contexte) pour choisir entre shift et réduction plutôt que la restriction de cet ensemble au contexte dans lequel on se trouve.

L'algorithme $LALR(1)$ (Lookahead $LR(1)$) rassemble certains item qui ne diffèrent que par le symbole lu en avant (item 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.

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 dérécursiver une grammaire récursive à gauche.
- Les parseurs ascendants manipulent des item : 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 essayent d'identifier successivement les réductions candidates (*handle*). Il y de nombreuses techniques pour trouver les réductions candidates. Les parseurs $LR(1)$ utilisent des ensembles d'item $LR(1)$.
- Un ensemble d'item 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'item.
- 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 symbole (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 réductions.

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 besoin de plus d'information que l'on ne peut pas coder avec une grammaire hors contexte, on va avoir besoin d'information *contextuelle*. Par exemple, lorsqu'on rencontre un identificateur, est-ce une fonction ou une variable ; si c'est une variable, quel type de valeur est stockée dans cette variable ; 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. En plus de reconnaître si le programme est bien une phrase du langage, le parseur va construire un contexte qui va lui permettre d'avoir accès à ces informations en temps voulu.

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. 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 connait 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 les nombres : entiers de différentes tailles, réels (flottants), caractères et booléens (quelquefois les chaîne de caractères).

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 : $type1 \times type2$ (souvent appelé structure) permett d'associer plusieurs objets de type arbitraire, les unions permettent de rassembler plusieurs types en un seul. 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 (difficile à gérer pour les gros programmes) ou lorsqu'ils ont la même structure (peut réduire les erreurs détectées par le système de typage).

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.

Remarque : Dans le cas des langages fortement typés, l'inférence de type se fait relativement bien. Il existe des situations plus difficiles. Par exemple, dans certains langages les déclarations sont optionnelles, mais le typage doit être consistant quand même (le type ne change pas d'une utilisation à l'autre). On introduit des algorithmes d'unification pour trouver le type le plus large compatible avec l'utilisation qui est faite des variables. D'autres langages permettent le changement dynamique de 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 :

$$T = \{+, -, 0, 1\}\ NT = \{Number, Sign, List, Bit\}\ S = \{Number\}\ P = \left\{ \begin{array}{lcl} Number & \rightarrow & Sign\ \ List \\ Sign & \rightarrow & + \\ & | & - \\ List & \rightarrow & List\ \ Bit \\ & | & Bit \\ Bit & \rightarrow & 0 \\ & | & 1 \end{array} \right\}$$

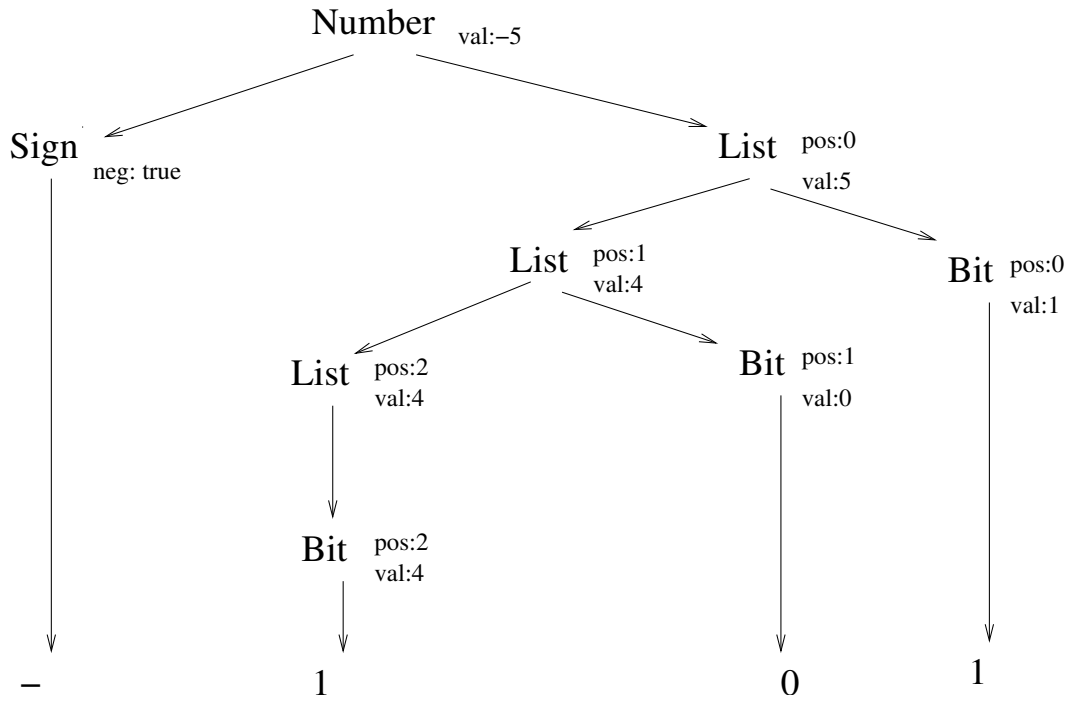
On va annoter cette grammaire avec des attributs permettant de calculer la valeur du nombre binaire signé qui est représenté :

Symbole	*Attribut*
$Number$	$valeur$
$Sign$	$negatif$
$List$	$position, valeur$
Bit	$position, valeur$

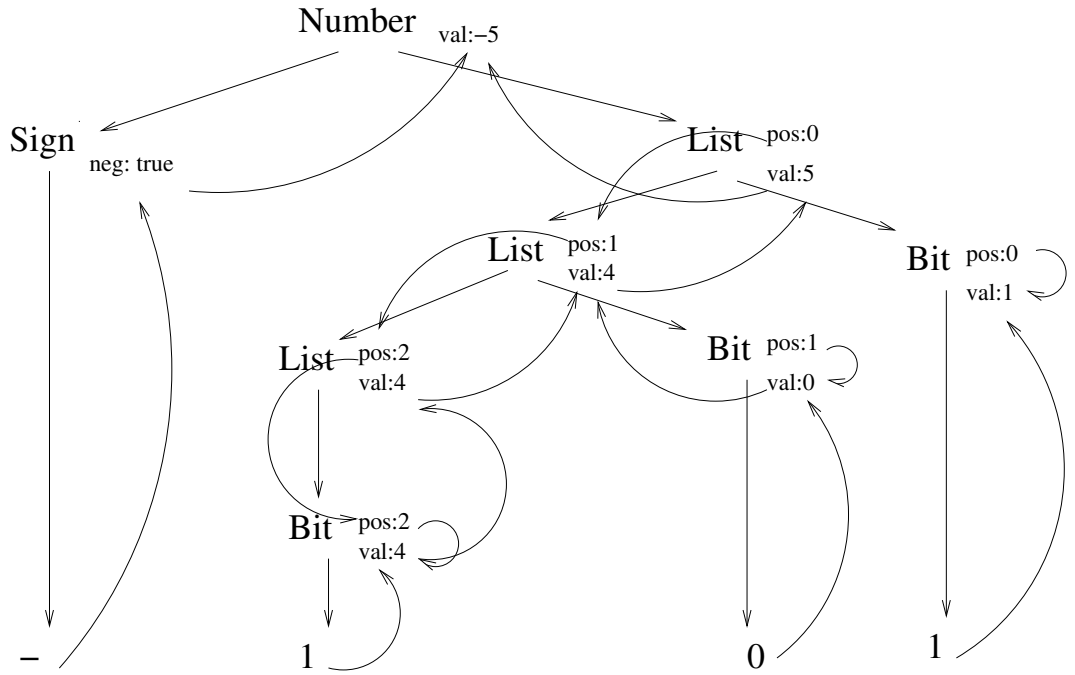
On va ensuite définir les règles pour définir les attributs de chaque symbole. Ces règles sont associées aux règles de la grammaire :

1.	$Number \rightarrow Sign\ \ List$	$List.position$	$\leftarrow 0$
		$si\ \ Sign.negatif$	
		$\quad alors\ \ Number.valeur$	$\leftarrow -List.valeur$
		$\quad sinon\ \ Number.valeur$	$\leftarrow List.valeur$
2.	$Sign \rightarrow +$	$Sign.negatif$	$\leftarrow false$
3.	$Sign \rightarrow -$	$Sign.negatif$	$\leftarrow true$
4.	$List \rightarrow Bit$	$Bit.position$	$\leftarrow List.position$
		$List.valeur$	$\leftarrow Bit.valeur$
5.	$List_0 \rightarrow List_1\ \ Bit$	$List_1.position$	$\leftarrow List_0.position + 1$
		$Bit.position$	$\leftarrow List_0.position$
		$List_0.valeur$	$\leftarrow List_1.valeur + Bit.valeur$
6.	$Bit \rightarrow 0$	$Bit.valeur$	$\leftarrow 0$
7.	$Bit \rightarrow 1$	$Bit.valeur$	$\leftarrow 2^{Bit.position}$

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



On peut représenter les dépendences 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. 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.
- 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 à 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. Elles restent néanmoins un moyen relativement propre et efficace de calculer certaines valeurs lorsque la structure du calcul s'y prête bien.

4.3 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 synthetisé (calculé de bas en haut). Les actions sont spécifiées par le concepteur du compilateur et peuvent être des actions complexes référençant des variables globales (table des symbole par exemple).

Le concepteur du compilateur ajoute des morceaux de code qui seront exécutés lors de l'analyse syntaxique. Chaque bout de code ou *action* est directement attachée à une production dans la grammaire. 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. Dans un parseur $LR(1)$ les actions sont à réaliser lors des réductions.

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

1.	$Number$	$\rightarrow$	$Sign\ \ List$	$Number.val$	$\leftarrow$	$Sign.val \times List.val$
2.	$Sign$	$\rightarrow$	$+$	$Sign.val$	$\leftarrow$	1
3.	$Sign$	$\rightarrow$	$-$	$Sign.val$	$\leftarrow$	-1
4.	$List$	$\rightarrow$	Bit	$List.val$	$\leftarrow$	$Bit.val$
5.	$List_0$	$\rightarrow$	$List_1\ \ Bit$	$List_0.val$	$\leftarrow$	$2 \times List_1.val + Bit.val$
6.	Bit	$\rightarrow$	0	$Bit.val$	$\leftarrow$	0
7.	Bit	$\rightarrow$	1	$Bit.val$	$\leftarrow$	1

4.4 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 déja 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 emplacement des symboles de la partie droite de la règle sont représentés par \$1, \$2 ...$\$n$. On peut réécrire l'exemple précédent avec ces conventions :

1.	$Number$	$\rightarrow$	$Sign\ \ List$	\$\$	$\leftarrow$	\$1 $\times$ \$2
2.	$Sign$	$\rightarrow$	$+$	\$\$	$\leftarrow$	1
3.	$Sign$	$\rightarrow$	$-$	\$\$	$\leftarrow$	-1
4.	$List$	$\rightarrow$	Bit	\$\$	$\leftarrow$	\$1
5.	$List_0$	$\rightarrow$	$List_1\ \ Bit$	\$\$	$\leftarrow$	$2 \times$ \$1 + \$2
6.	Bit	$\rightarrow$	0	\$\$	$\leftarrow$	0
7.	Bit	$\rightarrow$	1	\$\$	$\leftarrow$	1

Lorsque le programmeur veut effectuer une action entre l'évaluation de deux symbole en partie droite d'une règle, il peut ajouter une règle, il peut aussi modifier le code du parser lorsqu'il réalise un shift.

On a plus de souplesse au niveau du traitement, en particulier on peut décider de construire des structures de données globales accessibles depuis n'importe quelle règle. C'est par exemple le cas pour la table des symboles (souvent implementée comme une table hachage) qui permet d'accèder rapidement aux informations contextuelles concernant les identificateurs.

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

On dispose généralement d'une série de procédures $MakeNode_i(\ldots)$ qui construisent des noeuds ayant i fils. Lorsqu'on 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 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 général on préfère une version compressée de l'arbre de syntaxe appelée arbre de syntaxe abstraite. Pour cela, on garde les éléments essentiels de l'arbre de syntaxe mais on enlève les noeuds internes qui ne servent à rien. En pratique :

- 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.

Autre exemple : au lieu de générer un AST à partir d'une expression arithmétique, on génère directement du code assembleur ILOC. On suppose que l'on dispose des procédures $NextRegister$ qui retourne un nom de registre non utilisé, $Adress$ qui retourne l'adresse d'une variable en mémoire, $Value$ pour les constantes et $Emit$ qui génère le code correspondant à l'opération.

5 Représentations Intermédiaires

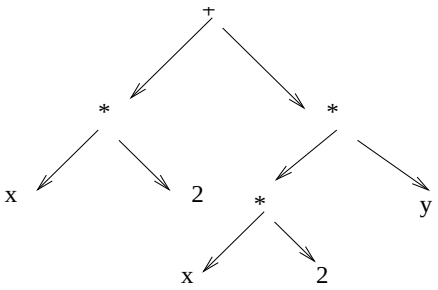
La structure de données utilisée pour effectuer les différentes passes de compilation après le parsing est appelée la représentation intermédiaire. En fait il y aura plusieurs représentations intermédiaires pour les différentes passes du compilateur. On distingue deux grandes classes de représentations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

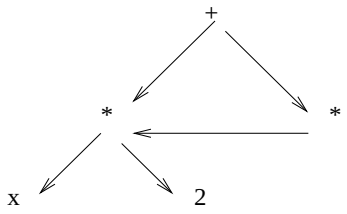
- Les représentations intermédiaires à base de graphes. Les algorithmes travaillant dessus manipulent des noeuds, des arcs, des listes, des arbres, etc. Les arbres de syntaxe abstraite construits au chapitre précédent sont de ce type.
- Les représentations intermédiaires linéaires qui ressemblent à du 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 contrôle flot). Ces représentations influencent fortement la manière dont le concepteur du compilateur va programmer ses algorithmes.

5.1 Graphes

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 (DAG) en regroupant les noeuds équivalents. Par exemple, on peut représenter l'expression $x \times 2 + x \times 2 \times y$ par :





DAG

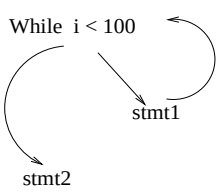
AST

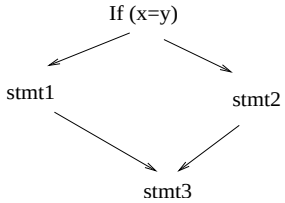
Ce type d'optimisation doit être fait avec précaution ; on doit être sûr que la valeur de x ne change pas entre les deux utilisations, on peut utiliser un système de table de hachage basé sur la syntaxe externe des morceaux de code. On utilise cette optimisation d'une part parce qu'elle réduit la place utilisée mais aussi parce qu'elle expose des redondances potentielles. Notons que l'on utilise aussi de telles représentations intermédiaires pour des représentations bas niveau

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

While $(i < 100)$
begin
$stmt_1$
end
$stmt_2$

If $(x = y)$
then $stmt_1$
else $stmt_2$
$stmt_3$

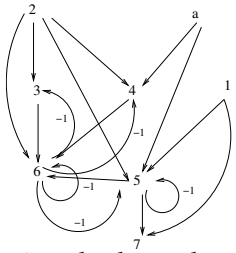




Ce graphe permet de suivre l'exécution qui aura lieu beaucoup mieux qu'avec l'AST. On a le choix de la granularité, c'est à dire de ce que contient un bloc (un noeud du graphe). En général on utilise les noeuds pour représenter les *blocs de base* (basic blocs). Les blocs de base représentent des portions de code dans lesquelles 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. Les blocs de base sont représentés en utilisant une autre représentation intermédiaire : AST ou code linéaire trois adresses.

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. graphe de dépendance permet de 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 à 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 de tableau 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

Beaucoup de compilateurs utilisent des représentations intermédiaires linéaires. Avant l'arrivée des compilateurs, c'était la manière standard pour programmer. Historiquement, on a utilisé des codes une adresse (machine à pile), codes deux adresses ou codes trois adresses.

Code une adresse 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 pendant un moment. 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). Ils sont utilisés pour les interpréteurs bytecode (Java, smalltalk)

code deux adresses Les codes deux adresses autorisent les opérations du type $x \leftarrow x \ \underline{op} \ y$ (on écrit $x \leftarrow \underline{op} \ y$). 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.

$push$	2	t_1	$\leftarrow$	2	t_1	$\leftarrow$	2
$push$	y	t_2	$\leftarrow$	y	t_2	$\leftarrow$	y
$multiply$		t_1	$\leftarrow$	$\times t_2$	t_3	$\leftarrow$	$t_1 \times t_2$
$push$	x	t_3	$\leftarrow$	x	t_4	$\leftarrow$	x
$substract$		t_3	$\leftarrow$	$-t_1$	t_5	$\leftarrow$	$t_4 - t_3$

5.2.1 code trois adresses

Les codes trois adresses autorisent les instructions du type : $x \leftarrow \ y \ \underline{op} \ z$ 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. repésentation 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_1	Arg_2
t_1		2	
t_2		y	
t_3	$\times$	(1)	(2)
t_4		x	
t_5	$-$	(4)	(3)

Op	Arg_1	Arg_2
	2	
	y	
$\times$	(1)	(2)
	x	
$-$	(4)	(3)

$Instr$
(1)
(2)
(3)
(4)
(5)

Op	Arg_1	Arg_2
	2	
	y	
$\times$	(1)	(2)
	x	
$-$	(4)	(3)

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 réfère une définition.

Pour transformer un code linéaire en SSA, le compilateur introduit ce qu'on appelle des $\phi - fonctions$ et renomme les variables pour créer l'assignation unique. Considérons l'exemple suivant :

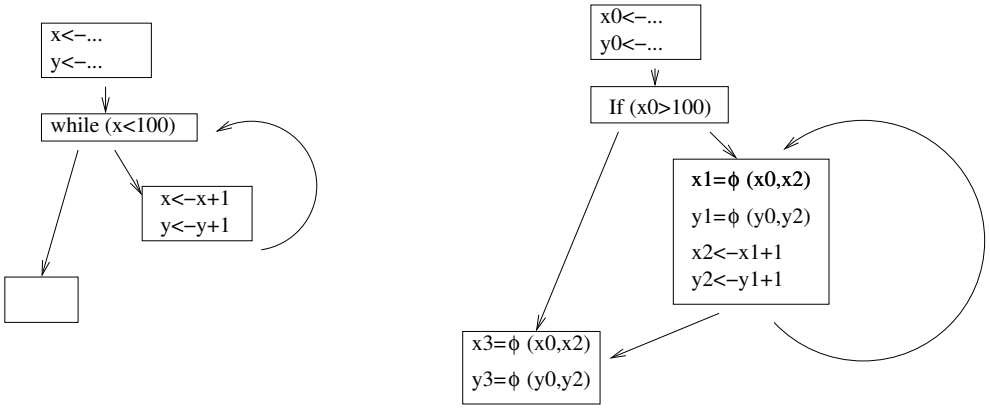
$x \leftarrow ...$
$y \leftarrow ...$
while $(x < 100)$
    $x \leftarrow x + 1$
    $y \leftarrow y + x$
end

Le passage en SSA va donner :

    $x_0 \leftarrow ...$
    $y_0 \leftarrow ...$
    if $(x_0 \geq 100)$ goto next
loop : $x_1 \leftarrow \phi(x_0, x_2)$
    $y_1 \leftarrow \phi(y_0, y_2)$

$x_2 \leftarrow x_1 + 1$
$y_2 \leftarrow y_1 + x_2$
if $(x_2 < 100)$ goto loop
next : $x_3 \leftarrow \phi(x_0, x_2)$
$y_3 \leftarrow \phi(y_0, y_2)$

Notons que le while a disparu,la forme SSA est une représentation utilisant le graphe de flot de contrôle ; la conservation du while impliquerait de rajouter des blocs ce que l'on évite de faire. On voit mieux ce qui se passe si on dessine le CDFG



On utilise fréquemment la relation de dominance entre instructions : a domine b signifie : "tout chemin arrivant a 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 ϕ, alors la définition de x domine s.
3. Si x est le $ieme$ argument d'une fonction ϕ dans le bloc n, alors la définition de x domine la dernière instruction du $ieme$ prédécesseur de n.

Les fonctions ϕ ne sont jamais implémentées dans le code généré, elle peuvent être supprimées en introduisant des copies.

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 (par exemple si on a une instruction $case$).

5.4 Nommage des valeurs et modèle mémoire

Une des sources d'optimisation du compilateur vient du fait que dans sa représentation intermédiaire, il nomme beaucoup plus de valeurs que ne le fait un langage de haut niveau. Par exemple, sur un simple appel de tableau $A[i, j]$, le codage dans un IR linéaire trois adresses du type ILOC sera :

$load$	1	$\Rightarrow$	r_1	
sub	r_j, r_1	$\Rightarrow$	r_2	$//r_2 \leftarrow j - 1$
$loadI$	10	$\Rightarrow$	r_3	// le tableau A est rangé par ligne, il des ligne de taille 10
$mult$	r_2, r_3	$\Rightarrow$	r_4	
sub	r_i, r_1	$\Rightarrow$	r_5	
add	r_4, r_5	$\Rightarrow$	r_6	$//r_6 \leftarrow 10 \times (j-1) + i - 1$
$loadI$	$@A$	$\Rightarrow$	r_7	
add	r_7, r_6	$\Rightarrow$	r_8	
$load$	r_8	$\Rightarrow$	$r_{A_{ij}}$	$//r_{A_{ij}} \leftarrow @A + 10 \times (j-1) + i - 1$

On voit que beaucoup de valeurs sont renommées et peuvent être utilisées par la suite car les registres ne sont pas écrasés.

Cette implémentation d'un accès à un tableau est faite en choisissant un modèle de mémorisation du type *registre à registre*. Dans ce modèle de mémorisation (ou modèle mémoire), le compilateur conserve les valeurs dans des registres sans se soucier du nombre de registres disponibles effectivement sur la machine cible. Les valeurs peuvent n'être stockées en mémoire que lorsque la sémantique du programme l'impose (par exemple lors d'un appel de procédure où l'adresse d'une variable est passée en paramètre). On génère alors explicitement les instructions $load$ et $store$ nécessaires.

L'autre alternative est d'utiliser un modèle *mémoire à mémoire*. Le compilateur suppose alors que toutes les valeurs sont stockées en mémoire, les valeurs sont chargées dans des registres juste avant leur utilisation et remise en mémoire juste après. L'espace des noms est plus important. On peut aussi choisir d'introduire des opérations entre cases mémoire. La phase d'allocation de registre sera différente suivant le modèle mémoire choisi.

Pour les machines RISC on utilise plus fréquemment le modèle registre à registre qui est plus proche du mode de programmation de l'architecture cible. D'autre part, cela permet de conserver des informations au cours du processus de compilation du type "cette valeur n'a pas été changée depuis son calcul" et cela très facilement si il est en SSA

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.

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 $h(nom)$)
- $inserer(nom, info)$ pour insérer l'information $info$ à l'adresse $h(nom)$. 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 :

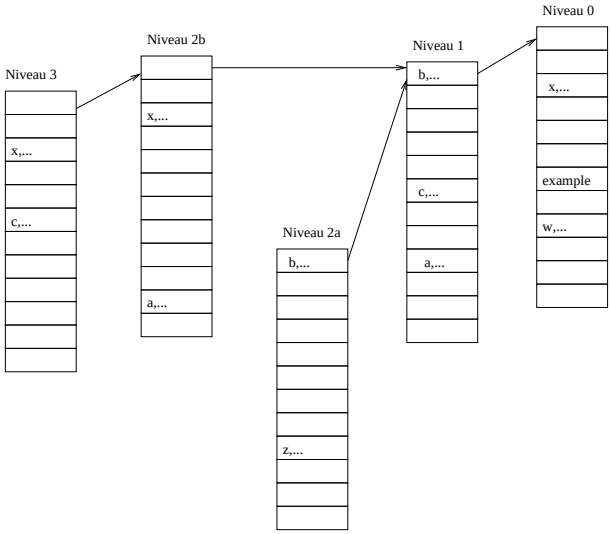
```
static int w;          /* niveau 0 */
int x;

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 $\langle niveau, decalage \rangle$ ou $niveau$ est le niveau lexical dans lequel la définition de la variable référencée a lieu et $decalage$ 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.

6 Procédures

Les procédures sont les unités de base pour tous les compilateurs, grâce à la compilation séparée. 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. La référence aux paramètres se fait indépendamment de leur nom à l'extérieur. Cet espace est alloué à l'appel de la procédure et désalloué 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 librairies 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. Nous verrons un peu les nouveautés avec la programmation objet.

6.1 Abstraction du contrôle et espace des noms

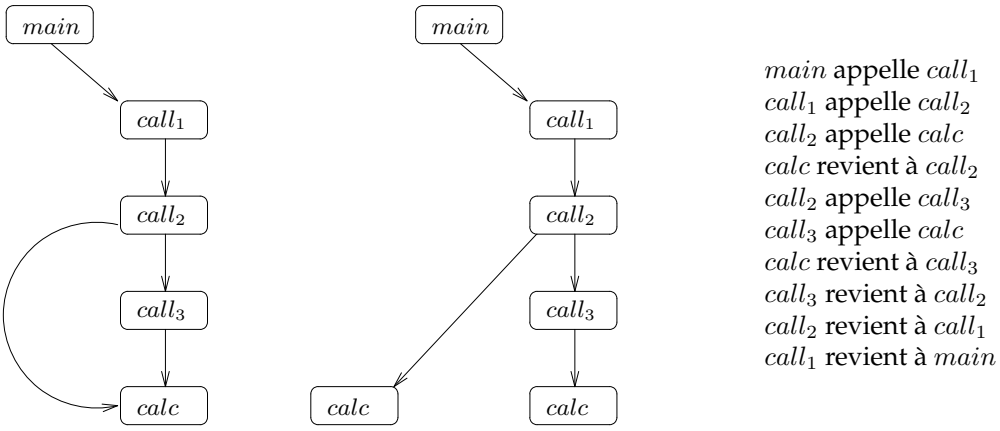
Le mécanisme de transfert de contrôle entre les procédures est le suivant : lorsqu'on appelle une procédure, le contrôle est donné à la procédure appelée ; lorsque cette procédure appelée se termine, le contrôle est redonné à la procédure appelante, juste après l'appel. Deux appels à une même procédure créent donc deux *instances* (ou invocations) indépendantes. On peut visualiser cela en représentant l'arbre d'appel 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 call_1;
        var y...
        procedure call_2
            var z : ...
            procedure call_3;
                var y....
                begin { call_3}
                    x :=...
                    calc;
                end;
            begin { call_2}
                z :=1;
                calc;
                call_3;
            end;
        begin { call_1}
            call_2;
            ...
        end;
```

```
begin { main }
    ...
    call_1 ;
end ;
```

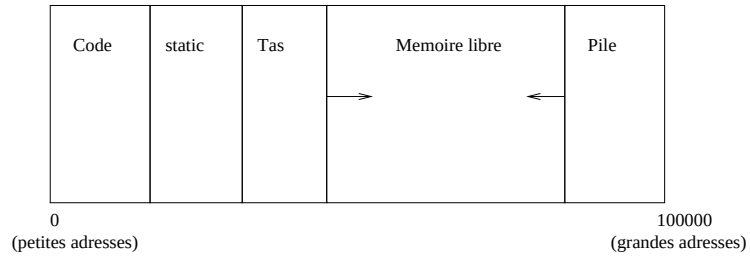
la différence entre le graphe d'appel (*call graph*) et l'arbre d'appel (qui est la version déroulée du graphe d'appel correspondant à une exécution particulière). L'historique d'exécution est un parcours de l'arbre d'exécution.



Ce mode d'appel et de retour s'exécute bien avec une pile. Lorsque $call_1$ appelle $call_2$, il pousse l'adresse de retour sur la pile. Lorsque $call_2$ finit, il dépile l'adresse de retour et branche à cette adresse. Ce mécanisme fonctionne aussi avec les appels récursifs. Certains ordinateurs (Vax-11) ont cablé ces accès à la pile dans les instructions d'appel et de retour de procédure.

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.

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 les appels de procédures (elle grandit vers les petites adresses), le tas sert à allouer la mémoire dynamiquement (il grandit vers les grandes adresses).

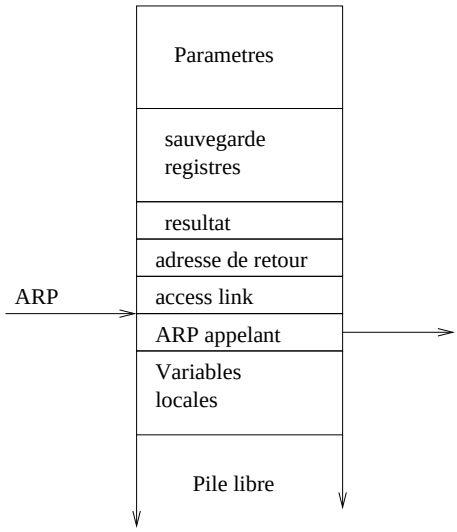


6.2 Enregistrement d'activation (AR)

Lors de l'appel d'une procédure, le compilateur met en place une région de la mémoire appelée *Enregistrement d'activation* (*activation record* : AR) qui va implémenter le mécanisme des appels et retours de la procédure. Cette zone mémoire est en général allouée sur la pile. Sauf cas exceptionnel, elle est désallouée lorsqu'on sort de la procédure.

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)

ainsi que d'autres variables pour contrôler l'exécution. En particulier, il contient 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.



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 est de taille dépendant de l'appel).

Variables locales. Lors de l'appel de la procédure, le compilateur calcule la taille des variables locales, référence pour chaque variable le niveau d'imbrication et le décalage par rapport à l'ARP. Il enregistre ces informations dans la table des symboles. La variable peut alors être accédée par l'instructions :

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

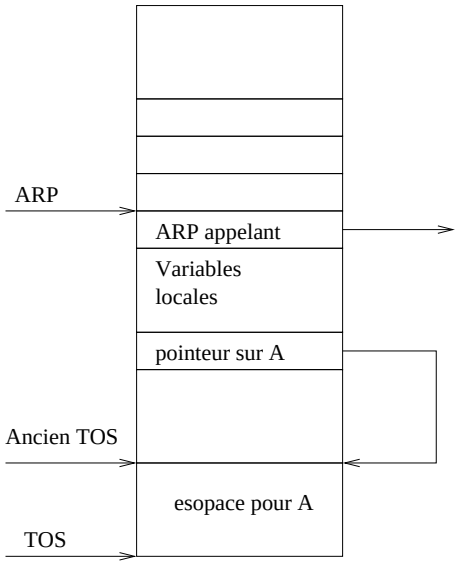
Lorsque la taille de la variable ne peut pas être connue statiquement (provenant d'une entrée du programme), le compilateur alloue un pointeur sur une zone mémoire qui sera allouée sur le tas (*heap*) lorsque sa taille sera connue. Le compilateur doit alors générer un code d'accès légèrement différent, l'adresse de la variable est récupérée avec $loadA0$, puis un deuxième load permet d'accéder à la variable. 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. Pour une variable statique (dont la durée de vie est indépendante de la procédure), le compilateur peut insérer la valeur directement au moment de l'édition de lien. 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).

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.

Allocation de l'enregistrement d'activation Le compilateur a plusieurs choix, en général la procédure appelante ne peut pas allouer complètement l'enregistrement d'activation, il lui manque des données comme la taille des variables locales.

Lorsque la durée de vie de l'AR est limitée à celle de la procédure, le compilateur peut allouer sur la pile. Le surcoût de l'allocation et de la désallocation est faible. On peut aussi implémenter des données de tailles variables sur la pile, en modifiant dynamiquement la fin de l'AR.



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 ou 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.

Dans certain 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 deux modes principaux de passage des paramètres entre les procédures : passage par valeur et passage par référence. Considérerons l'exemple C suivant :

```
int proc(x, y)                    c = proc(2, 3);
    int x, y;                     a = 2;
{                                 b = 3;
    x = 2 * x;                    c = proc(a, b);
    y = x + y;                    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 :

appel par valeur	*a in*	*a out*	*b in*	*b out*	*resultat*
$proc(2,3)$	–	–	–	–	7
$proc(a,b)$	2	2	3	3	7
$proc(a,a)$	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)                    c = proc(2, 3)
    returns fixed binary                a = 2 ;
    declare x,y fixed binary ;
    begin                               b = 3 ;
        x = 2 * x;                      c = proc(a, b)
        y = x + y;                      a = 2
        return y ;                      b = 3;
    end                                 c = proc(a, a);
```

et le résultat de l'exécution :

appel par valeur	*a in*	*a out*	*b in*	*b out*	*resultat*
$proc(2,3)$	–	–	–	–	7
$proc(a,b)$	2	4	3	7	7
$proc(a,a)$	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).

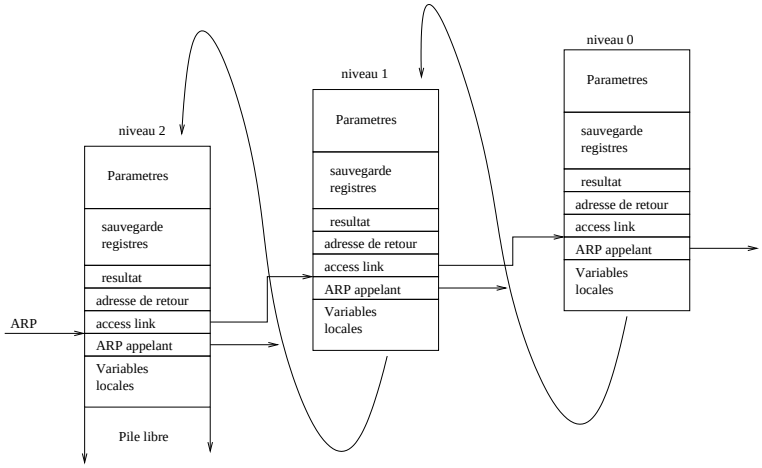
La valeur retournée par la procédure doit être en général stockée en dehors de l'AR car elle est utilisée alors que la procédure a terminé. L'AR inclut une place pour le résultat de la procédure. La procédure appelante doit alors elle aussi allouer de la place pour le résultat dans son propre AR et stocker 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 le décalage pour retrouver la variable) puis le décalage. 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. Plusieurs solutions sont possibles.Considérons par exemple que la prodédure $calc$ 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.

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.



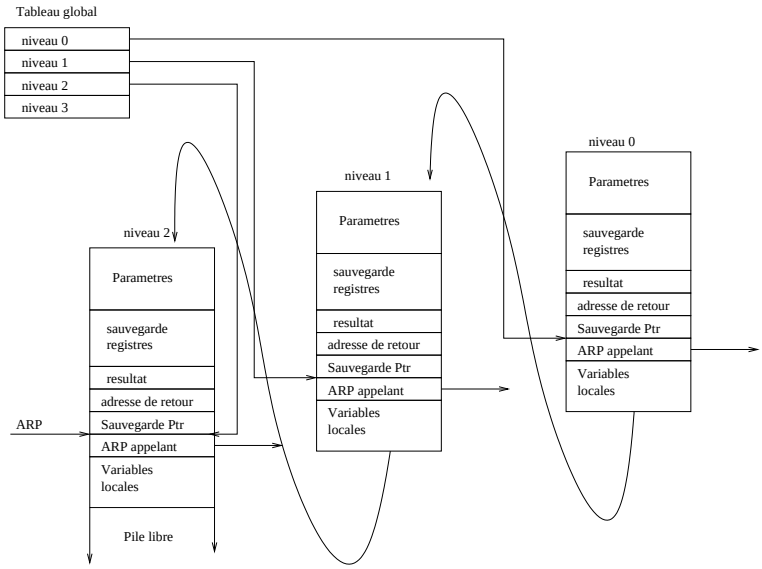
Par exemple, en supposant que le lien d'accès soit stocké avec un décalage 4 par rapport à l'ARP. Voici le type de code pour différents accès à différents niveaux (notons que quelquefois on préfère accéder aux variables en utilisant le sommet de pile (P) mais le mécanisme est similaire, avec des décalages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

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 un 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'ARP 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 Édition de lien

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. 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 pre-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 retrour, 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 restore 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) :

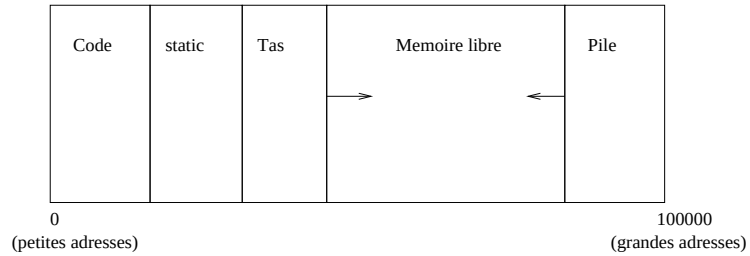
	appelé	appelant
appel	pre-appel	prologue
	alloue les bases de l'AR évalue et stocke les paramètres stocke l'adresse de retour et l'ARP sauve les registres (caller-save) saute à l'adresse appelée	préserve les registres (callee save) étend l'AR pour les données locales trouve l'espace des données statiques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

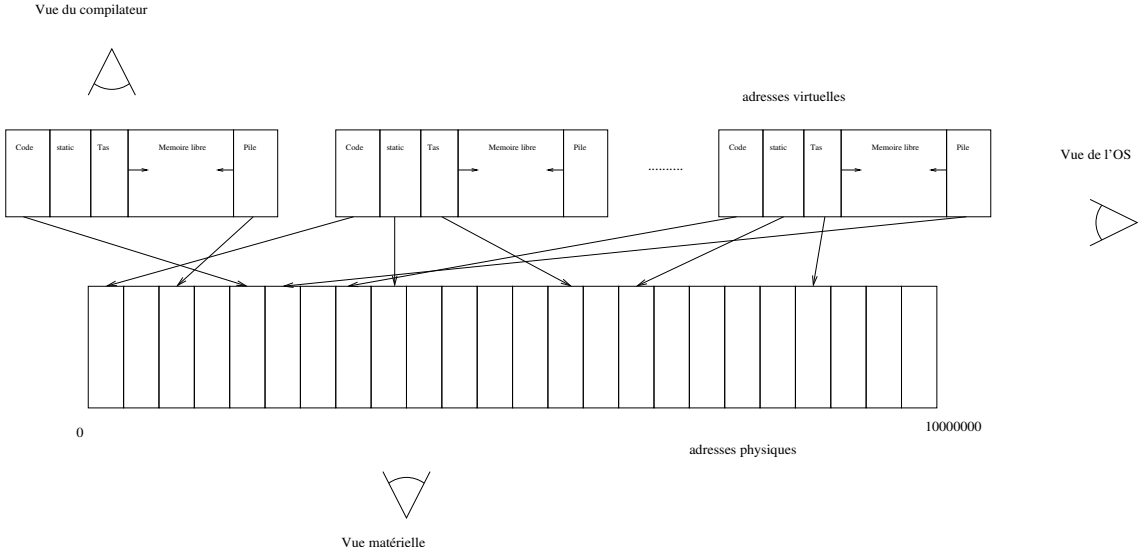
6.6 Gestion mémoire globale

Chaque programme comporte son propre adressage logique. L'exécution de plusieurs processus 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 correspondance 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 fonctionne 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'un 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 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 1, 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 succeptibles 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

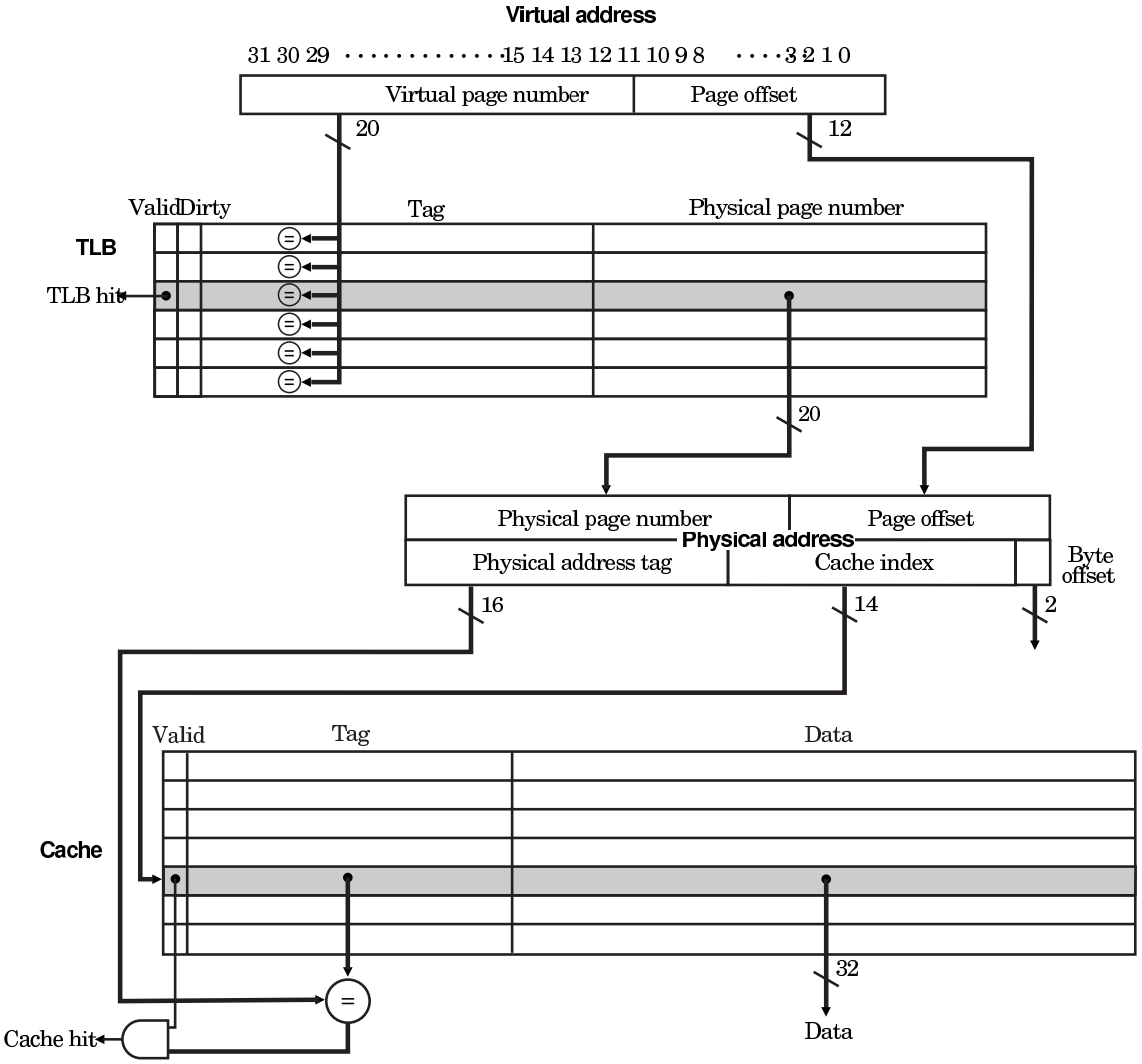


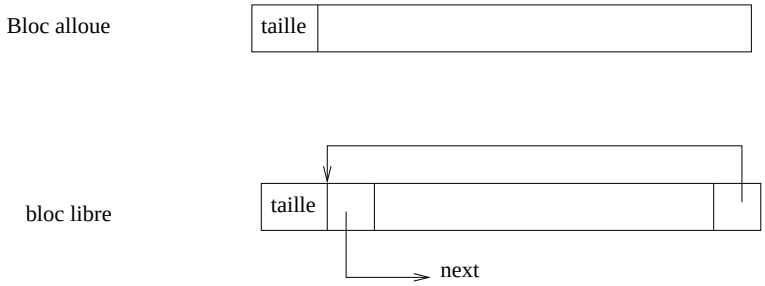
FIG. 1 – Implémentation du TLB et du cache sur les DECStation 3100 (figure tirée de [HP98]).

entre deux données n'est contôlable 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.

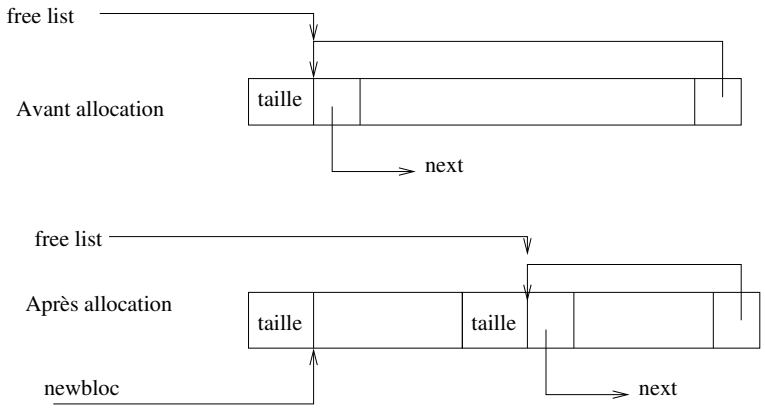
La méthode traditionnelle est appelée allocation du premier (*first-fit allocation*). Chaque bloc alloué a un champ caché qui contient sa taille (en général juste avant le champ retourné par la procédure d'allocation). Les blocs disponibles pour allocation sont accessibles par une liste dite liste libre (*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 suffissante.
- 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 précédant 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'agréger 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 couting*) 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
$WorkList \leftarrow$ { valeurs des pointeurs des AR et registres}
While ($WorkList \neq 0$)
    $p \leftarrow$tete de $WorkList$
    if ($p \rightarrow objet$ non marqué)
        marquer $p \rightarrow objet$
        ajouter les pointeurs de $p \rightarrow objet$ 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).

7 Implémentations de mécanismes spécifiques

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 (c.f. l'accès aux tableaux 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 concepteur 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 performance de l'analyse statique (ex : le mots clés `restric` 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 y a plusieurs classes de registres (ex : floatting point, general purpose). Les registres peuvent être distribués sur différents *register file* (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 représentation*) 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 :

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

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'l est utilisé comme un booléen dans une expression booléenne) est :

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

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

$$\begin{array}{ll} If & (x < y) \\ & then\ \ statement_1 \\ & else\ \ statement_2 \end{array}$$

à condition que le compilateur sache les reconnaître.

L'encodage par position peut paraître moins intuitif à 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 :

$$\begin{array}{lllll} & comp & r_a,\ r_b & \Rightarrow & cc_1 \\ & cbr\_LT & cc_1 & \rightarrow & L_3, L_1 \\ L_1: & comp & r_d,\ r_c & \Rightarrow & cc_2 \\ & cbr\_LT & cc_2 & \rightarrow & L_2, L_4 \\ L_2: & comp & r_e,\ r_f & \Rightarrow & cc_3 \\ & cbr\_LT & cc_3 & \rightarrow & L_3, L_4 \\ L_3: & loadI & true & \Rightarrow & r_1 \\ & jumpI & & \rightarrow & L_5 \\ L_4: & loadI & false & \Rightarrow & r_1 \\ & jumpI & & \rightarrow & L_5 \\ L5: & nop & & & \end{array}$$

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

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

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.

$$\begin{array}{lllll} & comp & r_a,\ r_b & \Rightarrow & cc_1 \\ & cbr\_LT & cc_1 & \rightarrow & L_1,\ L_2 \\ L_1: & cmp & r_c, r_d & \Rightarrow & cc_2 \\ & cbr\_LT & cc_2 & \rightarrow & L_3,\ L_2 \\ L_2 & load & false & \Rightarrow & r_x \\ & jumpI & & \rightarrow & L_4 \\ L_3 & load & true & \Rightarrow & r_x \\ & jumpI & & \rightarrow & L_4 \\ L_4 & nop & & & \end{array}$$

– Move conditionnel. On possède une instruction du type

$$i2i\_LT \quad cc_i,\ r_a,\ r_b \ \Rightarrow r_i$$

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.

$$\begin{array}{llll} comp & r_a,\ r_b & \Rightarrow & cc_1 \\ i2i\_LT & cc_1,\ r_{true},\ r_{false} & \Rightarrow & r_1 \\ comp & r_c,\ r_d & \Rightarrow & cc_2 \\ i2i\_LT & cc_2,\ r_{true},\ r_{false} & \Rightarrow & r_2 \\ and & r_1,\ r_2 & \Rightarrow & r_x \end{array}$$

– Comparaison à valeurs booléenes. 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 uniformité entre les booléens et les valeurs relationnelles.

$$\begin{array}{llll} cmp\_LT & r_a,\ r_b & \Rightarrow & r_1 \\ cmp\_LT & r_c,\ r_d & \Rightarrow & r_2 \\ and & r_1,\ r_2 & \Rightarrow & r_x \end{array}$$

– Exécution prédiquée. Certaines architectures autorisent le code prédiqué (ou prédicaté) 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 :

$$(r_1)?\ \ add\ r_1,\ r_2 \Rightarrow r_c$$

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.

$$\begin{array}{llll} cmp_LT & r_a,\ r_b & \Rightarrow & r_1 \\ cmp_LT & r_c,\ r_d & \Rightarrow & r_2 \\ and & r_1,\ r_2 & \Rightarrow & r_x \end{array}$$

7.3 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 paralléllisme. Par exemple, sur une machine pouvant exécuter deux instructions en parallèle :

$Unite1$	$unite2$
$comparaison \Rightarrow r_1$	
$(r_1)\ Op_1$	$\neg(r_1)\ Op_7$
$(r_1)\ Op_2$	$\neg(r_1)\ Op_8$
$(r_1)\ Op_3$	$\neg(r_1)\ Op_9$
$(r_1)\ Op_4$	$\neg(r_1)\ Op_{10}$
$(r_1)\ Op_5$	$\neg(r_1)\ Op_{11}$
$(r_1)\ Op_6$	$\neg(r_1)\ Op_{12}$

	$Unite1$	$unite2$
	$comp.andbranch.$	
$L1:$	Op_1	Op_2
	Op_3	Op_4
	Op_5	Op_6
	$jumpI\ \ L_{out}$	
$L2:$	Op_7	Op_8
	Op_9	Op_{10}
	Op_{11}	Op_{12}
	$jumpI\ \ L_{out}$	
$L_{out}:$	nop	

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) suit le schéma suivant :

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). On peut cacher cette latence en utilisant les *delay slot* (i.e. en exécutant des instructions *après* l'instruction de branchement pendant un ou deux cycles).

Exemple de génération de code pour une boucle for :

for $(i = 1; i <= 100; i++)$
{ corps de la boucle
}

	$loadI$	1	$\Rightarrow$	r_1
	$loadI$	1	$\Rightarrow$	r_2
	$loadI$	100	$\Rightarrow$	r_3
	cmp_LT	r_1, r_3	$\Rightarrow$	r_4
	cbr	r_4	$\rightarrow$	L_1, L_2
$L_1:$				
			$corps\ de\ la\ boucle$	
	add	r_1, r_2	$\Rightarrow$	r_1
	cmp_LT	r_1, r_3	$\Rightarrow$	r_6
	cbr	r_6	$\rightarrow$	L_1, L_2
$L_2:$			$suite$	

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)))
```

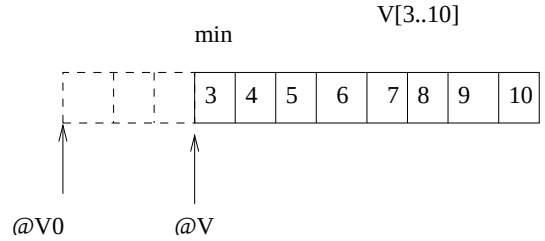
Lorsqu'on arrive à l'appel récursif, le résultat de l'appel en cours sera le résultat de l'appel récursif, tous les calculs de l'appel en cours ont été faits, on n'a donc plus besoin de l'AR de l'appel en cours. L'appel récursif peut être exécuté sans allouer un nouvel AR mais simplement en *recyclant* 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 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 tableau 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 $V[min \dots max]$. Pour accéder à $V[i]$, le compilateur devra calculer le décalage de cet élément du tableau par rapport à l'adresse de base à partir de laquelle il est stocké, le décalage est $(i - min) \times w$ ou w est la taille des éléments de w. Si $min = 0$ et w est une puissance de deux,le code généré s'en trouvera simplifié.

Si la valeur de 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 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 $V[i]$:

$loadI$	$@V_0$	$\Rightarrow$	r_1	// faux zéro pour V
$lshift$	$r_i, 2$	$\Rightarrow$	r_2	// $i \times$ taille des éléments
$loadA0$	r_1, r_2	$\Rightarrow$	r_2	// 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 :

1,1	1,2	1,3	1,4	2,1	2,2	2,3	2,4

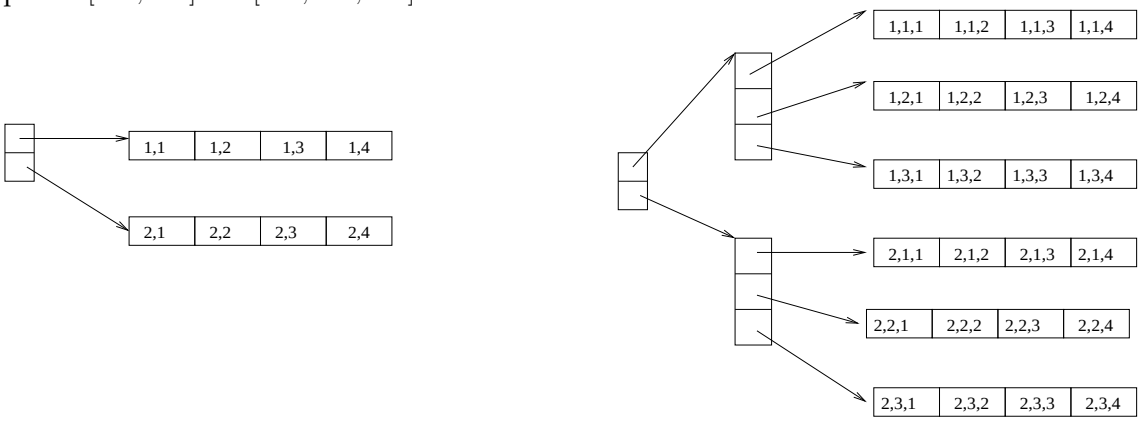
S'il est rangé par colonne on aura :

1,1	2,1	1,2	2,2	1,3	2,3	1,4	2,4

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 $i \leftarrow 1$ to 2
  for $j \leftarrow 1$ to 4
    $A[i, j] \leftarrow A[i, j] + 1$

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.

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 : $@A[i, j] = @A + i \times long_2 \times w + j \times w - (min_1 \times long_2 + min_2 \times w)$. 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.

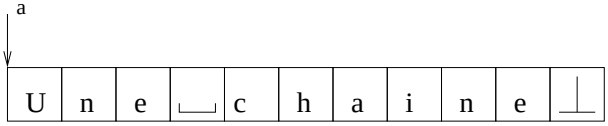
L'accès aux tableaux utilisant des pointeurs de vecteurs nécessite simplement deux instructions par dimension (chargement de la base de la dimension i, décalage jusqu'à l'élément dans la dimension i). Pour les machines où l'accès à la mémoire était rapide comparé aux opérations arithmétiques, cela valait 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.

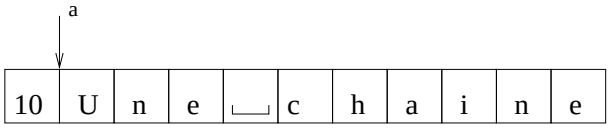
Beaucoup de compilateurs essaie de détecter les accès en dehors des bornes des tableaux. La méthode la plus simple est de faire un test sur les indices avant chaque appel à l'exécution. On peut faire un test moins précis en comparant le décalage calculé à la taille totale du tableau.

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 à un impact important sur les performance 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 ($cloadAI$, $cstoreAI$ par exemple pour Iloc). Par exemple pour traduire l'instruction C : $a[1] = b[2]$ où a et b sont des chaînes de caractères, on écrit :

$$\begin{array}{llll} loadI & @b & \Rightarrow & r_b \\ cloadAI & r_b, 2 & \Rightarrow & r_2 \\ loadI & @a & \Rightarrow & r_a \\ cstoreAI & r_2 & \Rightarrow & r_a, 1 \end{array}$$

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 :

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

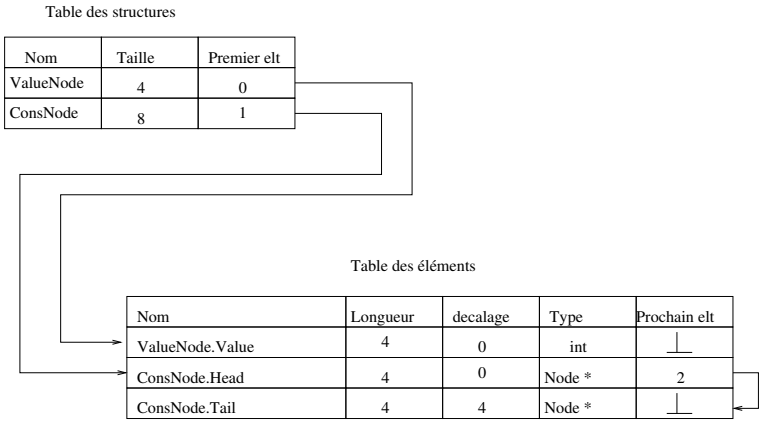
7.6 Structures

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
};

structu 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 $\rightarrow$ Head on peut générer l'instruction :

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

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.

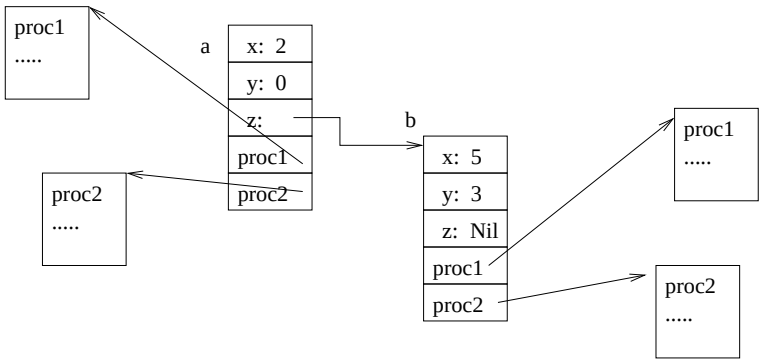
8 Implémentation des langages objet

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.

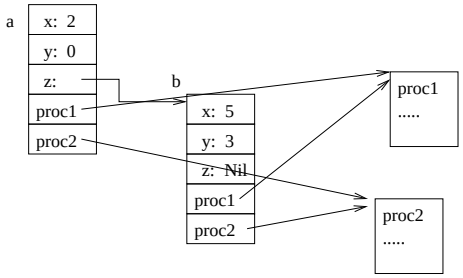
8.1 Espace des noms dans les langages objets

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 implementées comme dans les langages classiques).

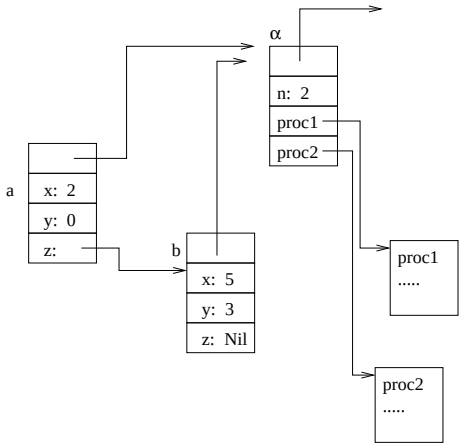
Pour le concepteur de compilateur, les objets nécessitent des mécanismes supplémentaires à la compilation et à l'exécution. Un objet est une abstraction qui possède plusieurs membres. Ces membres peuvent être des données, du code qui manipule ces données ou d'autres objets. On peut donc représenter un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

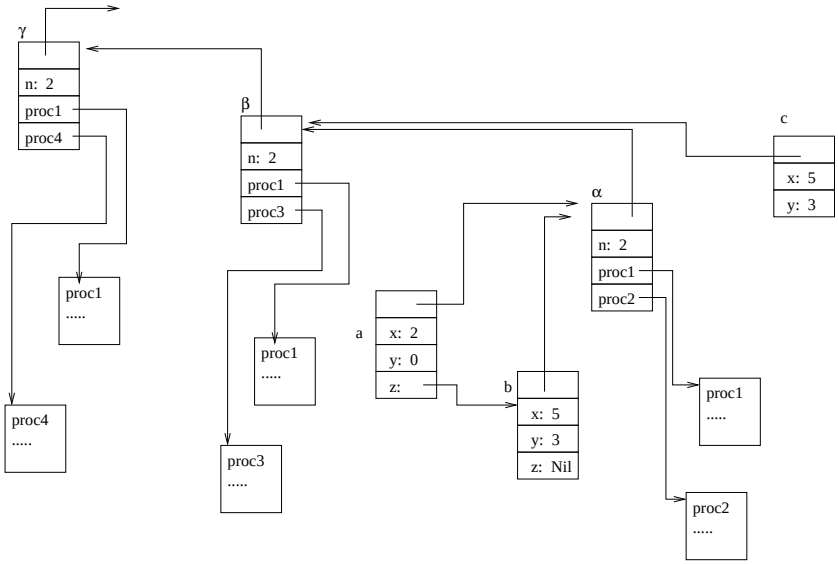


Ici α est la classe à laquelle appartiennent a et b. Les objets a et b ont un nouveau champ qui représente la classe à laquelle ils appartiennent (la classe étant un objet, elle a aussi ce nouveau champ). Attention, ces diagrammes sont des représentations de l'implementation de ces objets, pas de leur déclaration dans le langage source. Dans la déclaration de la classe α 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 implémentée 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 α peut avoir une classe parent β (ou super-classe). Toutes les méthodes de la classe β sont alors accessibles par un objet de la classe α. Toutes les variables d'instance de β sont variables d'instance de α 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 α dont β est une superclasse. Chacune des trois classes α, β et γ implémente une méthode nommée $proc_1$. On voit un objet c qui est une instance de la classe β, 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 α, s'il invoque $b.proc_4$ on exécutera la procédure définie dans la classe γ. Le mecanisme 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 α) 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 les 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 méthode).

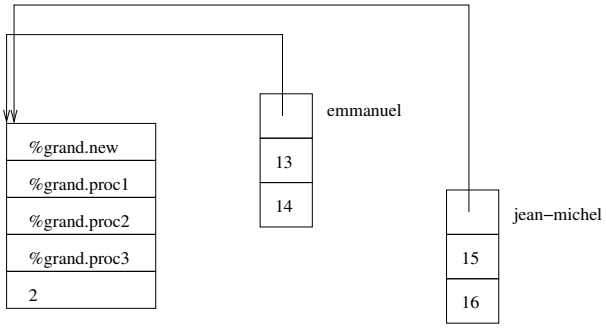
Dans ce cas, l'enregistrement d'objet pour une instance de la classe $grand$ est juste un vecteur de longueur 3. La première case est un pointeur sur sa classe ($grand$), qui contient l'adresse de la représentation concrète de la classe $grand$. Les deux autres cases contiennent les données x et y. Chaque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

	new	$proc_1$	$proc_2$	$proc_3$	n
$decalage$	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 :

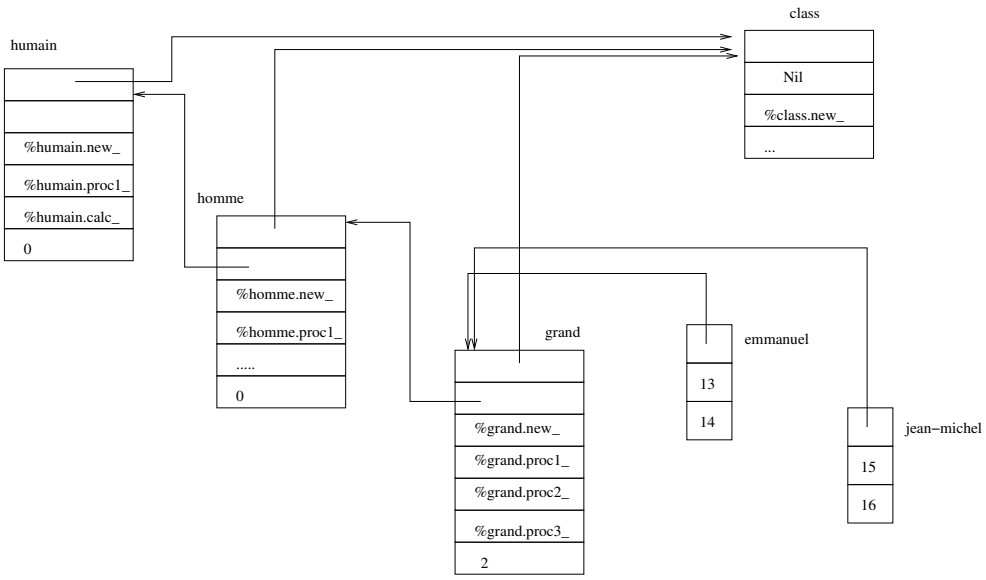


La table des méthodes de la classe $grand$ contient des labels (symbole créé par le compilateur). Ces labels sont associés aux adresses auxquelles sont stockées les procédures.

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 $jean-michel.proc_1$, 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 $proc_1$, 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 :

	new	$proc_1$	$proc_2$	$proc_3$	$calc$
$grand$	$\langle 0, 8\rangle$	$\langle 0, 12\rangle$	$\langle 0, 16\rangle$	$\langle 0, 20\rangle$	$\langle 2, 16\rangle$
$homme$	$\langle 0, 8\rangle$	$\langle 0, 12\rangle$	–	–	$\langle 1, 16\rangle$
$humain$	$\langle 0, 8\rangle$	$\langle 0, 12\rangle$	–	–	$\langle 0, 16\rangle$
$class$	$\langle 0, 8\rangle$	–	–	–	–

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 superclasse 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 enregistrement 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 :

$pointeurs\ de\ classes$	$data\ humain$	$data\ homme$	$data\ grand$

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 produire un arbre syntaxique, comment générer du code pour des structures particulières (structures de contrôle, appels de procédure, etc.). En général pour générer le code explicitement, 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 dinstructions, *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.

La difficulté du processus de sélection d'instructions vient du fait qu'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). Par exemple, pour copier de registre à registre on utilise naturellement l'opération Iloc :

$$i2i \quad r_i \quad \Rightarrow \quad r_j$$

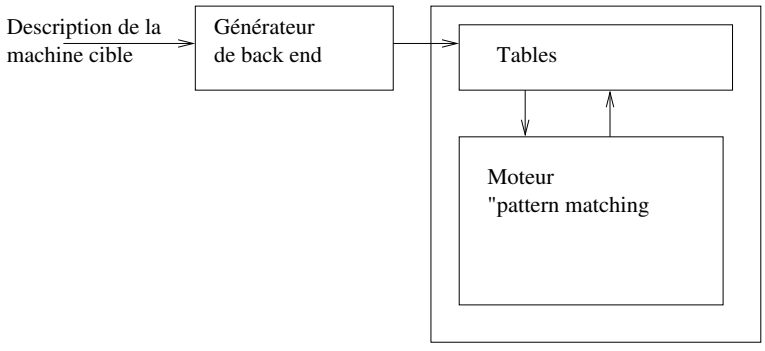
mais on peut aussi utiliser les opérations suivantes :

$addi$	$r_i, 0$	$\Rightarrow$	r_j	$subI$	$r_i, 0$	$\Rightarrow$	r_j
$multIi$	$r_i, 1$	$\Rightarrow$	r_j	$divI$	$r_i, 1$	$\Rightarrow$	r_j
$lshiftI$	$r_i, 0$	$\Rightarrow$	r_j	$rhiftI$	$r_i, 0$	$\Rightarrow$	r_j
orI	$r_i, 0$	$\Rightarrow$	r_j	$xorI$	$r_i, 0$	$\Rightarrow$	r_j

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. Certaines opérations (multiplication, division, accès mémoire) peuvent prendre plus d'un cycle d'horloge. Le temps des accès mémoire est difficilement prédictible (à cause du cache).

L'approche systématique de la sélection d'instructions permet de construire des compilateurs portables, c'est à dire qui peuvent cibler différentes machines sans avoir à réécrire la majeure partie du compilateur. Les trois opérations mentionnées plus haut opèrent sur le code assembleur de la machine cible, elles utilisent donc des informations spécifiques à la cible : par exemple, latence des opérations, taille des fichiers de registre, 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).

L'architecture d'un générateur automatique de sélecteur d'instructions est la 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

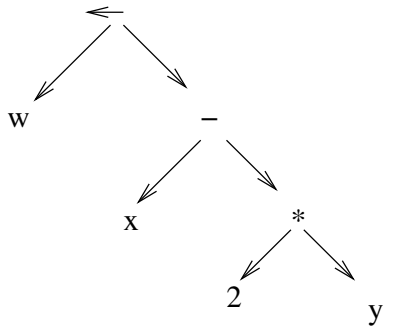
réécriture ascendant (*bottom up rewriting system*, BURS), il fonctionne sur les IR à base d'arbre. Le second utilise une optimisation classique : l'optimisation par fenêtrage (*peephole opimization*) qui essaie de trouver, parmi une librairie de morceaux de code prédéfinis (*pattern*), la meilleure manière d'implémenter un morceau de l'IR limité par une fenêtre (fonctionne sur les IR linéaires).

9.1 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.



op	$Arg1$	$Arg2$	$result$
$\times$	2	y	t_1
$-$	x	t_1	w

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(noeud){
    int result, t1, t2;
    switch(type(noeud))
    {
        case ×, −, +, ÷ :
            t1 ← expr(fils_gauche(noeud));
            t1 ← expr(fils_droit(noeud));
            result ← NextRegister();
            emit(op(noeud), t1, t2, result);
            break;
        case Identificateur :
            t1 ← base(noeud);
            t2 ← decalage(noeud);
            result ← NextRegister();
            emit(loadAI, t1, t2, result);
            break;
        case Number :
            result ← NextRegister();
            emit(loadI, val(noeud), result);
            break;
    }
return result
}
```

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

$$\begin{array}{llcl} loadAI & r_{arp}, @x & \Rightarrow & r_x \\ loadI & 2 & \Rightarrow & r_2 \\ loadAI & r_{arp}, @x & \Rightarrow & r_y \\ mult & r_2, r_y & \Rightarrow & r_3 \\ sub & r_x, r_3 & \Rightarrow & r_4 \end{array}$$

Mais pour avoir un code efficace il faudrait des traitements plus complexes sur les noeuds. 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 :

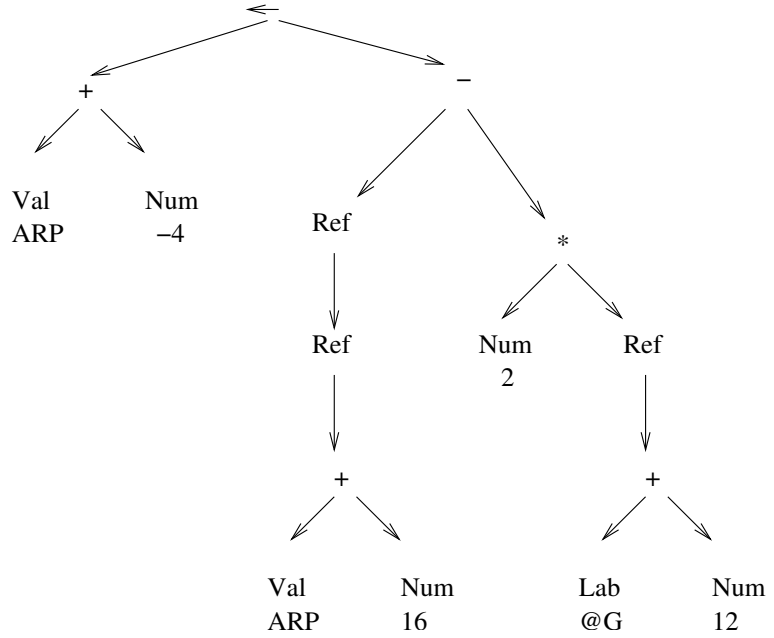
$$\begin{array}{llll} loadAI & r_{arp}, 4 & \Rightarrow & r_1 \\ loadI & 2 & \Rightarrow & r_2 \\ mult & r_1,\ r_2 & \Rightarrow & r_3 \end{array}$$

au lieu de

$$\begin{array}{llll} loadAI & r_{arp}, 4 & \Rightarrow & r_1 \\ multI & r_1,\ 2 & \Rightarrow & r_2 \end{array}$$

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 noeud (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.

Une solution utilisée est de raffiner la description de l'arbre en descendant à une description de niveau proche du code final. 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 noeud 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 définit une notation pour décrire les arbres en utilisant une notation préfixe. Par exemple l'arbre ci-dessus se décrira :

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

On dispose donc d'un ensemble d'*arbres d'opérations* (expression des instructions Iloc sous forme d'arbre) et l'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 $(noeud\ \ AST, arbre\ \ d'operation)$ qui indique que $arbre\ \ d'operation$ implémente $noeud\ \ AST$. Une fois que l'on a un pavage qui couvre tous les noeuds 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 noeuds 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 On va définir un certain nombre de règles de réécriture. Une règle de réécriture comporte une production exprimée sur la grammaire d'arbre, un code produit et un coût. Par exemple, une addition $+(r_1, r_2)$ correspond à la règle : $Reg \rightarrow +(Reg_1, Reg_2)$. 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 symbole Reg_1 et Reg_2 représentent aussi le nom terminal Reg, il sont numérotés pour pouvoir les identifier (comme pour les attributs). Le noeud $+$ décrit ainsi prend obligatoirement deux registres en argument et range son résultat dans un registre, il correspond a l'instruction Iloc add. 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). Considérons l'ensemble possible de règles suivantes pour réécrire l'AST de bas niveau mentionné précédemment.

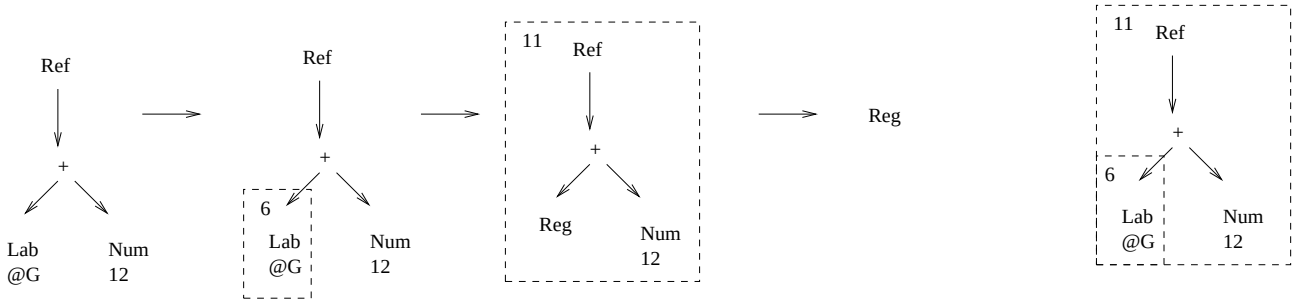
	règle			coût	code			
1	$Goal$	$\rightarrow$	$Assign$	0				
2	$Assign$	$\rightarrow$	$Gets(Reg_1, Reg_2)$	1	$store$	r_2	$\Rightarrow$	r_1
3	$Assign$	$\rightarrow$	$Gets(+(Reg_1, Reg_2), Reg_3)$	1	$storeAO$	r_3	$\Rightarrow$	r_1, r_2
4	$Assign$	$\rightarrow$	$Gets(+(Reg_1, Num_2), Reg_3)$	1	$storeAI$	r_3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

Pour comprendre comment utiliser ces règles, une notion importante est le *type de stockage* en mémoire 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* $Num \rightarrow Lab$). À chaque règle R_1, on peut associer un type de stockage pour la partie gauche et un type de stockage pour les arguments de la partie droite, pour l'algorithme mentionné plus loin, on appellera respectivement $left(R_1)$, $fils\_droit(right(R_1))$, $fils\_gauche(right(R_1))$ ces types de stockage. Par exemple, R_1 est la règle $Reg \rightarrow +(Num, Reg)$, $fils\_droit(right(R_1))$ vaut Num et $fils\_gauche(right(R_1))$ vaut Reg. Un ensemble de règle plus complet comprendra la distinction entre valeur stockée en registre et valeur stockée en mémoire.

La règle que nous avions 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 noeuds. 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
$loadA0$	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 ?

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 noeud de l'arbre un ensemble de règles qui peuvent être utilisées pour réécrire le noeud. On pourra alors par parcours en profondeur descendant (*pre-order*, ou *top down*) générer la meilleure instruction à chaque noeud.

Comme l'ensemble des configuration des noeuds de l'AST est finit, on va pouvoir générer statiquement l'ensemble des ensembles de règles qui pourrait s'appliquer sur un noeud d'un certain type, sans connaître le programme à compiler. On peut donc précalculer les transitions à effectuer à chaque noeud (comme pour un parseur $LL(1)$). Ceci permettra d'écrire un générateur de générateur de code (comme on vu qu'il existait des générateur de parseurs).

Définissons l'*étiquette* d'un noeud de l'AST comme un ensemble de règles de réécriture. Par exemple on a vu que le noeud $+_1$ 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 noeud $+_1$ 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 noeud $+_1$ (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 $Label(n)$ l'étiquette d'un noeud. Par extension on note $left(Label(n))$ l'ensemble des type de stockage des parties gauches des règles contenues dans l'étiquette (ici $left(Label(+_1)) = \{Reg\}$). Pour illustrer l'algorithme, supposons que toutes les opérations ont au plus deux arguments et que les parties droite des règles ne possèdent qu'une opération. Cela ne change rien au traitement car on peut toujours rajouter des non terminaux. On peut par exemple réécrire les règle 10, 11 et 12 en :

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 $Reg \rightarrow Ref(R12P2)$ $R12P2 \rightarrow +(Num, Reg)$

On choisit de laisser le coût total de chaque règle sur la première. Voici l'algorithme qui permet de calculer les étiquettes d'un AST.

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

En prenant toujours le même sous-AST : $Ref_1(+_1(Lab_1, Num_1))$, on peut exécuter l'algorithme. En arrivant sur le noeud Lab_1 les seules règles qui implémentent ce noeud sont les règles 6 et 22 : $Reg \rightarrow Lab$ et $Num \rightarrow Lab$. On construit donc l'étiquette de ce noeud par :

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

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

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

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

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

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

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

L'idée utilisée ici est la même pour pour le parsing $LL(1)$: on peut précalculer tous les états possibles. Il y en a un nombre fini. Si R est le nombre de règles, l'étiquette d'un noeud contient moins de R éléments donc il y en a moins de 2^R : $|ens\_label| < 2^R$ (bien sûr c'est toujours beaucoup moins que cela vu la structure de la grammaire). Si chaque règle a un opérateur et deux opérandes, on calcule complètement l'étiquette d'un noeud à partir de l'opération du noeud, de l'étiquette de son fils droit et de l'étiquette de son fils gauche. On peut précalculer les transitions d'une étiquette à une autre dans une table de taille :

$$|ens\_arbres\_operation| \times |ens\_label| \times |ens\_label|$$

Cette table est en général creuse et il existe des techniques efficaces pour la comprimer.

On a donc la possibilité de déterminer *avant la compilation*, l'automate qui va associer l'ensemble des règles que l'on pourra affecter à chaque noeud. On peut donc générer l'ensemble des générations de code possibles pour un AST donné. En associant à chaque génération de code son coût, on peut choisir la meilleure. Bien sûr cela représente un nombre exponentiel de possibilités, on va voir tout de suite comment faire cela plus efficacement grâce à un algorithme de type programmation dynamique.

On peut associer des coûts à chaque règle calculé à partir des coûts de chaque règle et des coûts des fils droits et gauches. On note $R_1@val_1$ pour indiquer que le choix de la règle R_1 pour le noeud coûtera val_1. Pour notre exemple cela donne :

$$Label(Num_1) = \{Num@0, Reg \rightarrow Num@1\}$$

$$Label(Lab_1) = \{Lab@0, Num \rightarrow Lab@0, Reg \rightarrow Lab@1\}$$

$$Label(+_1) = \left\{ \begin{array}{l} Reg \rightarrow +(Reg_1, Reg_2)@3, Reg \rightarrow +(Reg_1, Num_2)@2, Reg \rightarrow +(Num_1, Reg_2)@2, \\ R10P2 \rightarrow +(Reg, Reg)@2, R11P2 \rightarrow +(Reg, Num)@1, R12P2 \rightarrow +(Num, Reg)@1, \end{array} \right\}$$

$$Label(Ref_1) = \left\{ \begin{array}{l} Reg \rightarrow Ref(Reg_1)@3, Reg \rightarrow Ref(R10P2)@3, \\ Reg \rightarrow Ref(R11P2)@2, Reg \rightarrow Ref(R12P2)@2 \end{array} \right\}$$

Avec ce système un noeud contient l'information du coût total du sous arbre pour chaque règle pouvant implémenter le noeud. Ici on a effectué une première sélection en ne conservant que les noeuds 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 noeud on ne conserve que les noeuds les moins chers pour un stockage donné, on obtient :

$$Label(Num_1) = \{Num@0, Reg \rightarrow Num@1\}$$

$$Label(Lab_1) = \{Lab@0, Num \rightarrow Lab@0, Reg \rightarrow Lab@1\}$$

$$Label(+_1) = \left\{ \begin{array}{l} Reg \rightarrow +(Reg_1, Num_2)@2, Reg \rightarrow +(Num_1, Reg_2)@2, \\ R10P2 \rightarrow +(Reg, Reg)@2, R11P2 \rightarrow +(Reg, Num)@1, R12P2 \rightarrow +(Num, Reg)@1, \end{array} \right\}$$

$$Label(Ref_1) = \left\{ \begin{array}{l} Reg \rightarrow Ref(R11P2)@2, Reg \rightarrow Ref(R12P2)@2 \end{array} \right\}$$

Dans notre exemple, on voit que l'on peut ne garder que deux pavages pour ce noeud Ref_1 : les pavages correspondant à l'application des règles $\langle 6, 11\rangle$ et $\langle 22, 8, 12\rangle$. Si l'on choisi 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 :

$$Label(Num_1) = \{Num@0\}$$

$$Label(Lab_1) = \{Reg \rightarrow Lab@1\}$$

$$Label(+_1) = \left\{ \begin{array}{l} R11P2 \rightarrow +(Reg, Num)@1 \end{array} \right\}$$

$$Label(Ref_1) = \left\{ \begin{array}{l} Reg \rightarrow Ref(R11P2)@3 \end{array} \right\}$$

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 noeud, une passe *top down* qui choisit la règle à utiliser pour chaque noeud en fonction du meilleur coût pour le noeud racine et enfin un nouvelle

passe *bottom up* pour générer le code. Dans cette approche, les étiquettes sont calculées au moment de la compilation. Si maintenant on décide de précalculer ces nouvelles étiquettes, on ne va plus avoir un ensemble fini d'étiquette, les coût pouvant grossir arbitrairement. Si l'on désire utiliser cette approche (pour écrire un générateur de générateur de code), on va *normaliser* les étiquettes. En effet, l'information statique de chaque noeud est "combien le choix de telle règle rajoute au coût déjà calculé". Lorsqu'on va évaluer les coûts associés à chaque règle dans une étiquette, on va normaliser en retranchant le coût de la règle la moins chère à chacun des coûts. Pour notre exemple, cela ne change rien pour les noeuds Num_1 et Lab_1 mais pour les autres, on obtiendrait :

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

$$Label(Ref_1) = \left\{ \; Reg \rightarrow Ref(R11P2)@0, Reg \rightarrow Ref(R12P2)@0 \; \right\}$$

On peut faire cela systématiquement pour toutes les possibilités comme on l'a fait précédemment, mais cette fois en ne gardant que les règles qui on 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 noeud avec l'état. On peut aussi, calculer au vol le coût total associé à chaque noeud. Ensuite, le processus est le même que précédemment. C'est la technique dite "BURS" (*bottom up rewriting system*).

Plusieurs variantes de cette technique existent :

- On peut écrire à la main le programme de réécriture d'arbre (*pattern macher*), un peu comme l'algorithme $Tile$ ci dessus. Cela permet des optimisations et évite de manipuler une table qui peut être grosse (le générateur de code est compact).
- On peut utiliser la technique BURS vue ci-dessus, ce qui permet de faire de la compilation reciblable.
- On peut aussi utiliser des techniques encore plus proche du parsing, en étendant les techniques vues en début de cours pour fonctionner sur les grammaires fortement ambiguës utilisées ici.
- Enfin 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.

9.2 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 traitement très rapides.

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

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

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

$$\begin{array}{llll} addI & r_2, 0 & \Rightarrow & r_7 \\ mult & r_4, r_7 & \Rightarrow & r_{10} \end{array} \quad \Rightarrow \quad mult \quad r_4, r_2 \quad \Rightarrow \quad r_{10}$$

$$\begin{array}{rlll} & jumpI & \rightarrow & L_{10} \\ L_{10}: & jumpI & \rightarrow & L_{11} \end{array} \Rightarrow \begin{array}{rlll} & jumpI & \rightarrow & L_{11} \\ L_{10}: & jumpI & \rightarrow & L_{11} \end{array}$$

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 effet 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 $w \leftarrow x - \times y$:

op	$Arg1$	$Arg2$	$result$
$\times$	2	y	t_1
$-$	x	t_1	w

$\Rightarrow$

$$\begin{array}{rcl} r_{10} & \leftarrow & 2 \\ r_{11} & \leftarrow & @y \\ r_{12} & \leftarrow & r_{arp} + r_{11} \\ r_{13} & \leftarrow & M(r_{12}) \\ r_{14} & \leftarrow & r_{10} \times r_{13} \\ r_{15} & \leftarrow & @x \\ r_{16} & \leftarrow & r_{arp} + r_{15} \\ r_{17} & \leftarrow & M(r_{16}) \\ r_{18} & \leftarrow & r_{17} - r_{14} \\ r_{19} & \leftarrow & @w \\ r_{20} & \leftarrow & r_{arp} + r_{19} \\ M(r_{20}) & \leftarrow & r_{18} \end{array}$$

Supposons une de fenêtre 3 instructions :

$$\begin{array}{rcl} r_{10} & \leftarrow & 2 \\ r_{11} & \leftarrow & @y \\ r_{12} & \leftarrow & r_{arp} + r_{11} \end{array}$$
étape 1

$\Rightarrow$

$$\begin{array}{rcl} r_{10} & \leftarrow & 2 \\ r_{12} & \leftarrow & r_{arp} + @y \\ r_{13} & \leftarrow & M(r_{12}) \end{array}$$
étape 2

$\Rightarrow$

$$\begin{array}{rcl} r_{10} & \leftarrow & 2 \\ r_{13} & \leftarrow & M(r_{arp} + @y) \\ r_{14} & \leftarrow & r_{10} \times r_{13} \end{array}$$
étape 3

$\Rightarrow$

$$\begin{array}{rcl} r_{13} & \leftarrow & M(r_{arp} + @y) \\ r_{14} & \leftarrow & 2 \times r_{13} \\ r_{15} & \leftarrow & @x \end{array}$$
étape 4

$\Rightarrow$

$$\begin{array}{rcl} r_{14} & \leftarrow & 2 \times r_{13} \\ r_{15} & \leftarrow & @x \\ r_{16} & \leftarrow & r_{arp} + r_{15} \end{array}$$
étape 5

$\Rightarrow$

$$\begin{array}{rcl} r_{14} & \leftarrow & 2 \times r_{13} \\ r_{16} & \leftarrow & r_{arp} + @x \\ r_{17} & \leftarrow & M(r_{16}) \end{array}$$
étape 6

$\Rightarrow$

$$\begin{array}{rcl} r_{14} & \leftarrow & 2 \times r_{13} \\ r_{17} & \leftarrow & M(r_{arp} + @x) \\ r_{18} & \leftarrow & r_{17} - r_{14} \end{array}$$
étape 7

$\Rightarrow$

$$\begin{array}{rcl} r_{17} & \leftarrow & M(r_{arp} + @x) \\ r_{18} & \leftarrow & r_{17} - r_{14} \\ r_{19} & \leftarrow & @w \end{array}$$
étape 8

$\Rightarrow$

$$\begin{array}{rcl} r_{18} & \leftarrow & r_{17} - r_{14} \\ r_{19} & \leftarrow & @w \\ r_{20} & \leftarrow & r_{arp} + r_{19} \end{array}$$
étape 9

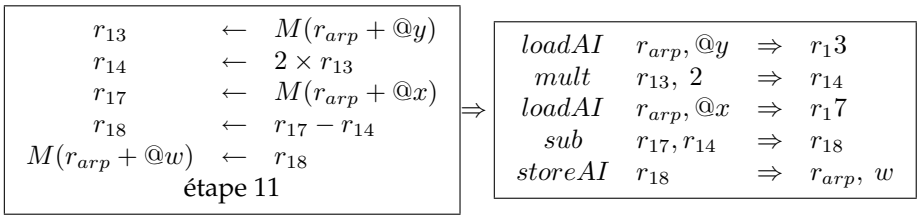
$\Rightarrow$

$$\begin{array}{rcl} r_{18} & \leftarrow & r_{17} - r_{14} \\ r_{20} & \leftarrow & r_{arp} + @w \\ M(r_{20}) & \leftarrow & r_{18} \end{array}$$
étape 10

$\Rightarrow$

$$\begin{array}{rcl} r_{18} & \leftarrow & r_{17} - r_{14} \\ M(r_{arp} + @w) & \leftarrow & r_{18} \end{array}$$
étape 11

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 définitions*). Une définition peut atteindre plusieurs premières utilisations et une utilisation peut correspondre à plusieurs définitions.

L'optimisation par fenêtrage a été beaucoup utilisé 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 recibler le code pour une grande variété d'architectures.

10 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 2 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.

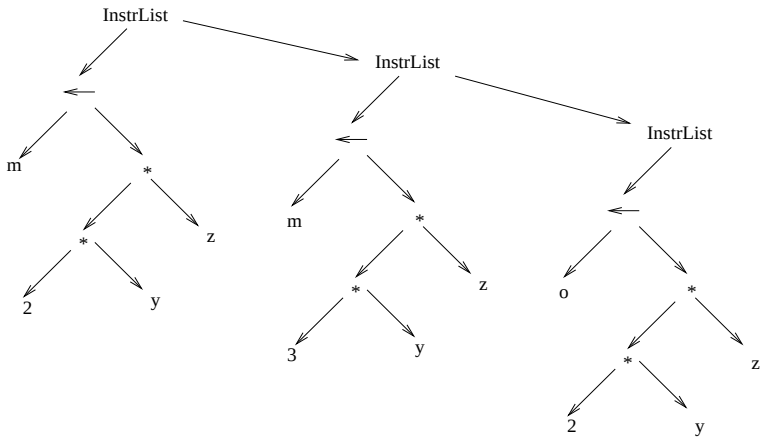
Plutôt que d'étudier beaucoup d'optimisations superficiellement, on va étudier une optimisation particulière : l'élimination d'expressions redondantes.

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.

$$\begin{array}{l} m \leftarrow 2 \times y \times z \\ n \leftarrow 3 \times y \times z \\ o \leftarrow 2 \times y - z \end{array} \qquad \begin{array}{l} t_0 \leftarrow 2 \times y \\ m \leftarrow t_0 \times z \\ n \leftarrow 3 \times y \times z \\ o \leftarrow t_0 - z \end{array}$$

10.1 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 :



On peut alors transformer cet arbre en un graphe acyclique (directed acyclic graph : DAG), dans

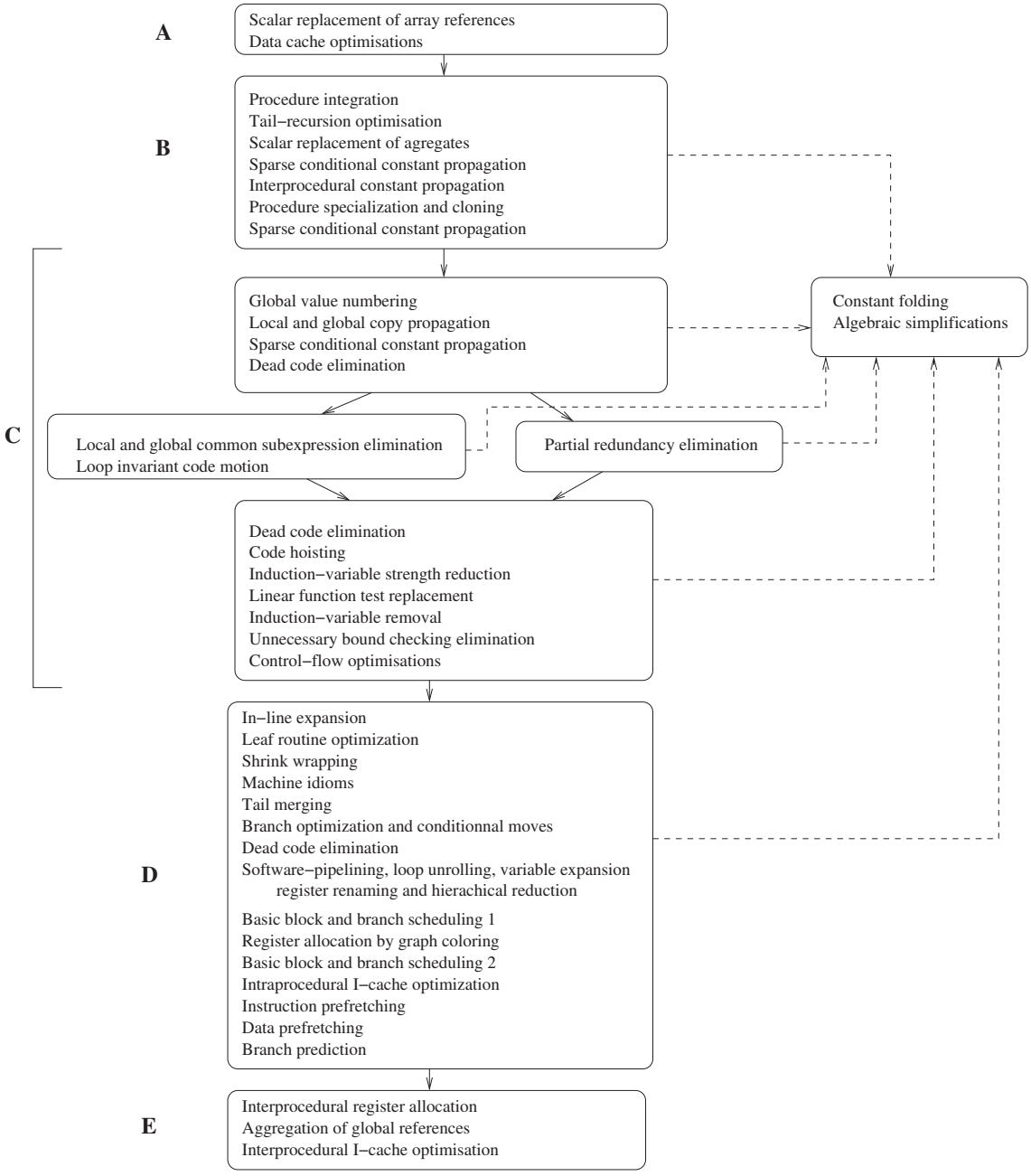
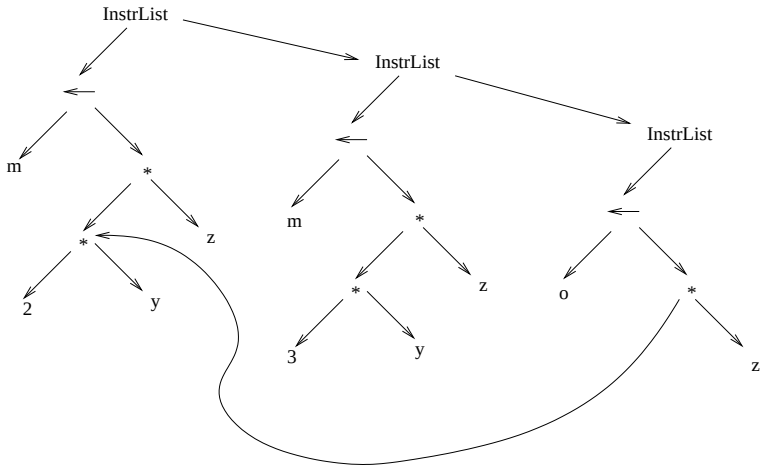


FIG. 2 – Ordre des optimisations d'après S. Muchnick

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.

10.2 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 opérandes ont les mêmes numéros de valeur. L'assignation de pointeur à 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 opérandes. 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 $op_i\ x_1,\ x_2, \ldots, x_{k-1} \Rightarrow x_k$
1) récupérer les numéros de valeurs de $x_1,\ x_2, \ldots, x_{k-1}$
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 $x_1,\ x_2, \ldots, x_{k-1}$
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 :

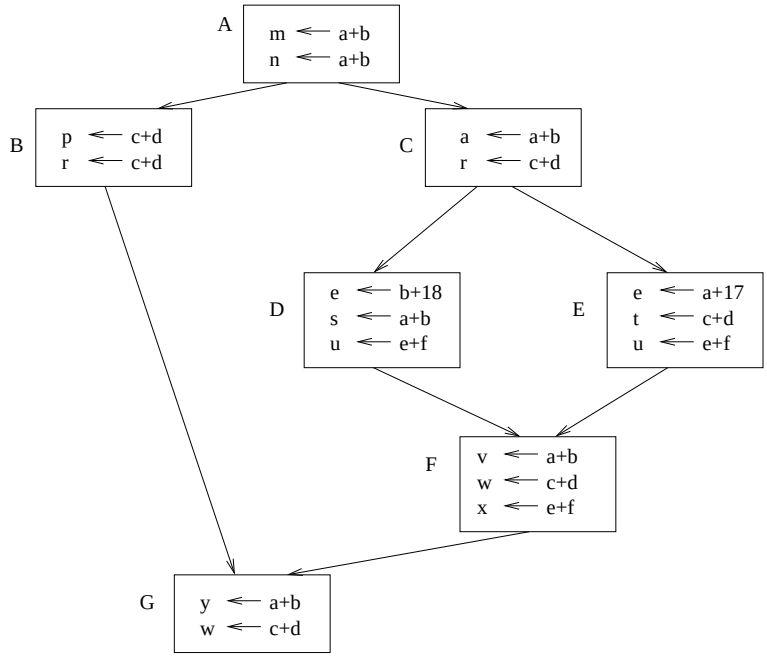
$a_0 \leftarrow x_0 + y_0$
$b_0 \leftarrow x_0 + y_0$
$a_1 \leftarrow 17$
$c_0 \leftarrow x_0 + y_0$
Code en SSA

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.

10.3 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 :

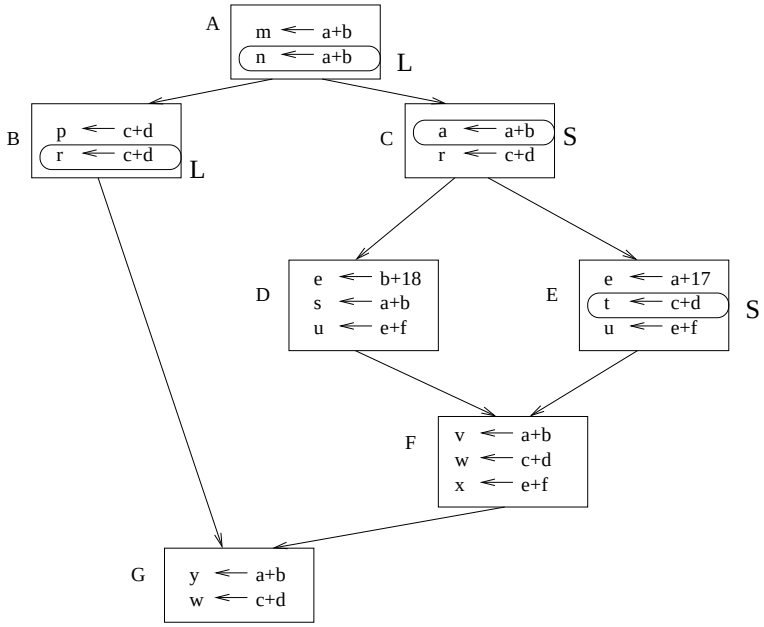


On distingue 4 types de méthodes :

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 $\beta_1, \beta_2, \ldots, \beta_n$ où β_1 a de multiples prédécesseurs et chaque β_i, $2 \leq i \leq n$ a β_{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.

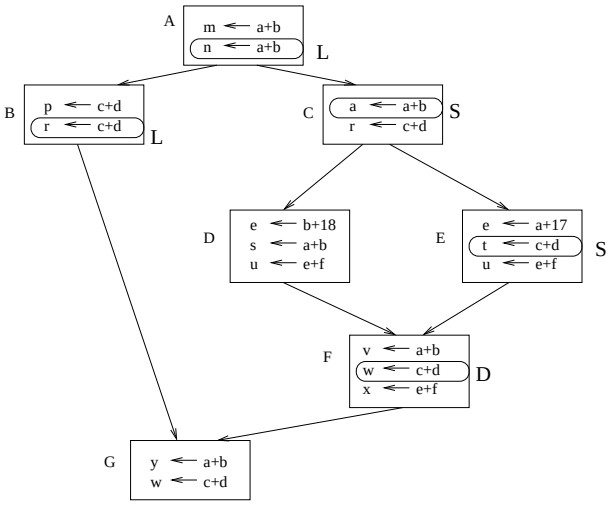
Approche super-locale On peut par exemple calculer la table de hachage correspondant au bloc A, puis l'utiliser comme état initial pour numéroter les valeurs du bloc C, puis utiliser la table résultante comme entrée pour le bloc D. Il faudra ensuite recalculer la table pour le bloc A afin de l'utiliser 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 ou est passé le contrôle. En revanche, on peut utilise 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 bloc 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 définit 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 $Avail(B)$:

$$Avail(B) = \bigcap_{p \in pred(B)} (Def(p) \cup (Avail(p) \cap NotKilled(p))) \quad (1)$$

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 $o_1, o_2, \ldots, o_k$.
$Killed \leftarrow \emptyset$
$Def(B) \leftarrow \emptyset$
for i=k to 1
    //supposons que o_i soit : "$x \leftarrow y + z$"
    si ($y \notin Killed$) et ($z \notin Killed$) alors
        ajouter "$x + y$" à $Def(B)$

ajouter x à $Killed$
$NotKilled(B) \leftarrow \{toutes\ les\ expressions\}$
Pour chaque expression e
    Pour chaque variables $v \in e$
        si $v \in Killed$ alors
            $NotKilled(B) \leftarrow NotKilled(B) - e$

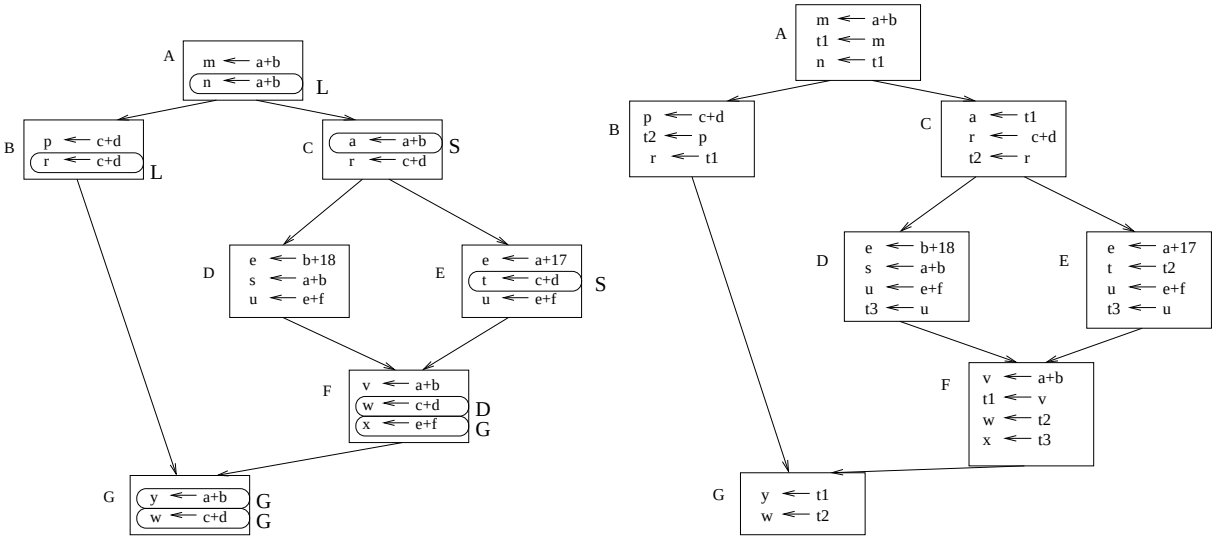
Pour calculer les ensembles $Avail$ on peut utiliser un simple algorithme itératif (on suppose que les blocs du programme ont été numérotés de b_0 à b_n).

for $0 \leq i \leq n$
    Calculer $Def(b_i)$ et $NotKilled(b_i)$
    $Avail(b_i) \leftarrow \emptyset$
$Changed \leftarrow True$
Tant que ($Changed$)
    $Changed \leftarrow False$
    for $0 \leq i \leq n$
        $OldValue \leftarrow Avail(b_i)$
        $Avail(b_i) := \bigcap_{p \in pred(b_i)} (Def(p) \cup (Avail(p) \cap NotKilled(p)))$
        si $Avail(b_i) \neq OldValue$ alors
            $Changed \leftarrow True$

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 (1) (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.

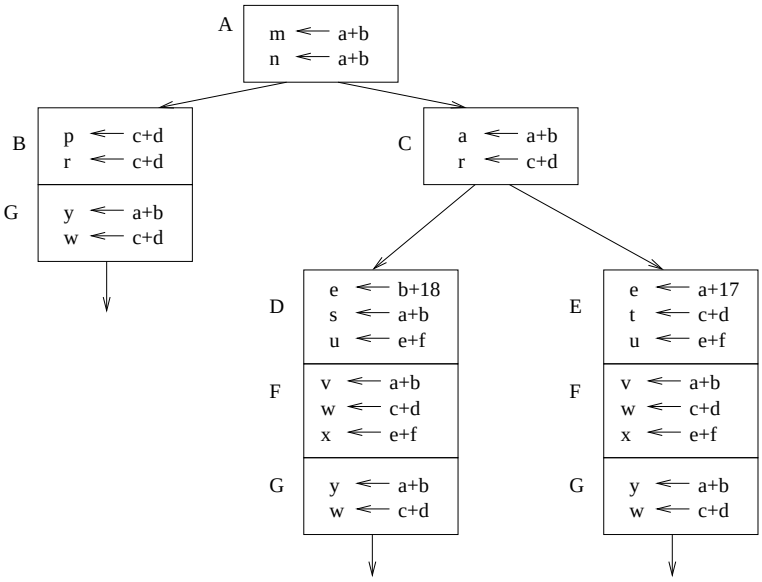
La phase de réécriture de l'élimination de redondances doit remplacer toute expression étiquetée comme redondante par une copie de la valeur calculée antérieurement (on suppose en général que ces copies peuvent être éliminées par une passe ultérieure d'optimisation).

Le traitement est effectué en deux passes linéaires sur le code : la première passe identifie les endroits ou 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 ou 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'existait 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.

11 Analyse flot de données

L'analyse flot de données (*data flow analysis*) est un raisonnement statique (à la compilation) sur les flots dynamiques des valeurs (à l'exécution). Cela consiste en la résolution d'un ensemble d'équations déduites d'une représentation graphique du programme pour découvrir ce qui peut arriver lorsque le programme est exécuté. On l'utilise essentiellement pour trouver des opportunités d'optimisation.

11.1 Exemple des variables vivantes

Une variable est *vivante* à un point p si il y a un chemin entre p et une utilisation de v suivant lequel v n'est pas redéfinie. Cette information est utilisée par de nombreuses optimisations (allocation de registres, passage en SSA, etc.).

L'ensemble $Live(b)$ des variables vivantes à la sortie du bloc b est défini par l'équation suivante :

$$Live(b) = \bigcup_{s \in succ(b)} Used(s) \cup (Live(s) \cap NotDef(s)) \tag{2}$$

Une variable est vivante à l'entrée d'un bloc s si elle est utilisée dans s ($\in Used(s)$) ou si elle est vivante en sortie de s sans avoir été redéfinie dans le s.

Ici, l'ensemble $Live$ d'un noeud est une fonction des ensembles $Live$ de ses successeurs, l'information remonte le long des arcs du graphe de flot de contrôle. On appelle cela un problème remontant (*backward problem*). $Avail$ était un problème descendant (*forward problem*).

Pour résoudre ce type de problème il faut construire le graphe de dépendance, rassembler des informations locales à chaque bloc (ici $Used$ et $NotDef$) puis utiliser un algorithme de point fixe itératif pour propager l'information dans le CFG.

Construction du graphe de flot de contrôle La construction du graphe de flot de contrôle se fait en général à partir d'une représentation intermédiaire linéaire. On identifie d'abord les instructions *leader* (qui sont en tête d'un bloc de base). Une instruction est leader si c'est la première de la procédure ou si elle possède un label qui est la cible potentielle d'un branchement. Si le programme ne possède par de saut sur registre ($jump \rightarrow r_1$ en Iloc), seul les labels cibles de branchements sont leader. Si il y a une telle instruction, tous les labels deviennent leader. Les compilateurs insèrent quelquefois des instructions virtuelles suivant les sauts qui indiquent les labels susceptibles d'être cibles du saut correspondant (instruction *tbl* en Iloc)

On identifie ensuite la fin des blocs, qui correspondent à un branchement conditionnel, un saut (il faut éventuellement prendre en compte les *delay slots*) ou un autre leader.

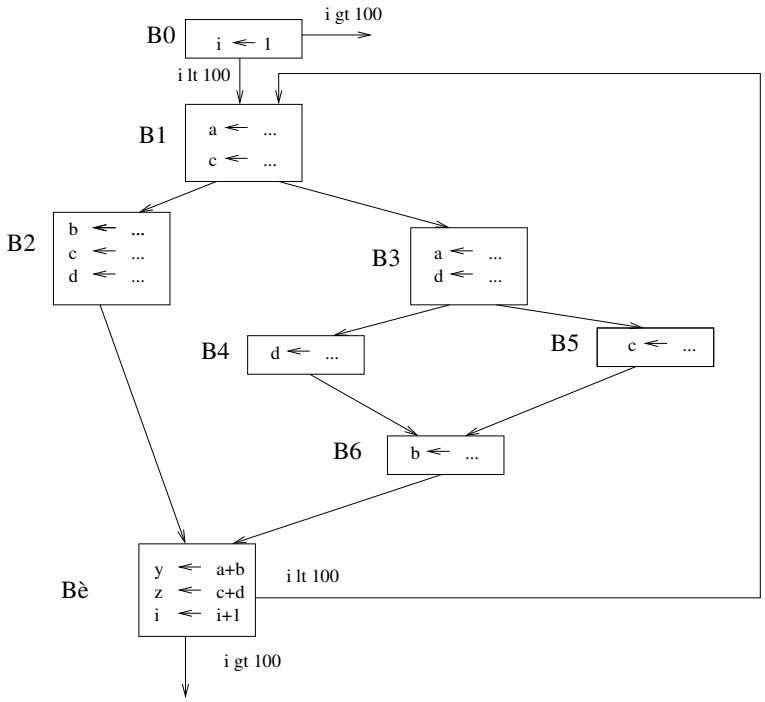
Informations locales aux blocs Le calcul des ensembles $Used$ et $NotDef$ est simple ici :

for i=1 to $\#operations$
  //supposons que o_i soit : "$op\ \ x, y \Rightarrow z$"
  si (o_i est leader) alors
    $b \leftarrow$ numéro de bloc de o_i
    $Used(b) \leftarrow \emptyset$
    $NotDef(b) \leftarrow$ {toutes les variables}
  si $x \in NotDef(b)$ alors
    ajouter x à $Used(b)$
  si $y \in NotDef(b)$ alors
    ajouter y à $Used(b)$
  $NotDef(b) \leftarrow NotDef(b) - z$

Résolution des équations Pour résoudre les équations on utilise une variante de l'algorithme vu pour la détection d'expressions redondantes.

for i=1 to $\#blocs$
    $Live(b_i) \leftarrow \emptyset$
$Changed \leftarrow True$
Tant que ($Changed$)
    $Changed \leftarrow False$
    for i=1 to $\#blocs$
        $OldValue \leftarrow Live(b_i)$
        recalculer $Live(b_i)$ en utilisant l'équation
        si $Live(b_i) \neq OldValue$ alors
            $Changed \leftarrow True$

Considérons le CFG suivant :



Le déroulement de ces algorithmes donne les résultats suivants :

	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7
$Used$	–	–	–	–	–	–	–	a, b, c, d, i
$NotDef$	a, b, c, d, y, z	b, d, i, y, z	a, i, y, z	b, c, i, y, z	a, b, c, i, y, z	a, b, d, i, y, z	a, c, d, i, y, z	a, b, c, d

itération	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7
0	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1	$\emptyset$	$\emptyset$	a, b, c, d, i	$\emptyset$	$\emptyset$	$\emptyset$	a, b, c, d, i	$\emptyset$
2	$\emptyset$	a, i	a, b, c, d, i	$\emptyset$	a, c, d, i	a, c, d, i	a, b, c, d, i	i
3	i	a, i	a, b, c, d, i	a, c, d, i	a, c, d, i	a, c, d, i	a, b, c, d, i	i
3	i	a, c, i	a, b, c, d, i	a, c, d, i	a, c, d, i	a, c, d, i	a, b, c, d, i	i
3	i	a, c, i	a, b, c, d, i	a, c, d, i	a, c, d, i	a, c, d, i	a, b, c, d, i	i

11.2 Formalisation

Nous allons introduire une formalisation pour la résolution de problèmes data flow par une méthode itérative (telle que celle que nous avons vu jusqu'à présent). On introduit la notion formelle de *problème data flow itératif* comme étant un quadruplet $\langle G, \mathcal{L}, \mathcal{F}, \mathcal{M} \rangle$ où :

- G est un graphe qui représente le flot de contrôle du morceau de code analysé (CFG dans une procédure, graphe d'appel dans un programme).
- $\mathcal{L}$ est un semi-treillis qui représente les faits analysés. Un semi-treillis est un triplet $\langle S, \wedge, \bot \rangle$. S est un ensemble, $\wedge$ est l'opérateur *minimum* (*meet* operator) et $\bot$ est l'élément minimal (*bottom*) de $\mathcal{L}$. On va associer un élément de S à chaque noeud de G.
- $\mathcal{F}$ est un espace de fonction :$\mathcal{F} : \mathcal{L} \rightarrow \mathcal{L}$. L'analyseur utilise les fonctions de $\mathcal{F}$ pour modéliser les transmissions de valeurs dans $\mathcal{L}$ le long des arcs G.
- $\mathcal{M}$ associe les arcs et les noeuds de G a certaines fonctions de $\mathcal{F}$.

Si $\mathcal{L}$ est un semi-treillis, alors l'opérateur $\wedge$ doit être idempotent ($a \wedge a = a$), commutative et associatif. Cet opérateur induit un ordre partiel sur $\mathcal{L}$: $a \geq b \Leftrightarrow a \wedge b = b$. Lorsque $a \wedge b = c$ avec $c \neq a$ et $c \neq b$, a et b ne sont pas comparable. L'élément minimal est l'unique élément $\bot$ tel que $a \wedge \bot = \bot$ (donc $a \geq \bot, \forall a \in \mathcal{L}$).

Pour la propriété de vivacité, on a la formalisation suivante :

- On recherche des ensembles de variables associés à chaque bloc, l'ensemble $\mathcal{S}$ peut donc être l'ensemble des parties de $\mathcal{V}$: $S = 2^{\mathcal{V}}$ ($\mathcal{V}$ étant l'ensemble des variables du programme).
- l'opérateur minimum doit spécifier comment combiner deux ensemble venant de deux flots de contrôle qui convergent, ici les ensembles sont réunis : $\wedge$ est l'union. La relation d'ordre partiel est alors : $a \leq b \Leftrightarrow b \subset a$.
- l'élément minimal est l'ensemble de toutes les variables $\bot = \mathcal{V}$

L'espace des fonctions doit permettre d'exprimer les opérations ensemblistes réalisées sur un bloc donné dans les équations data flow. Ici, on utilise uniquement la fonction : $f(x) = c_1 \cup (x \cap c_2)$ ($Used(b)$ et $NotDef(b)$ sont des constantes pour tout b). L'espace des fonctions est donc l'ensemble des fonctions de la forme $f(x) = c_1 \cup (x \cap c_2)$ pour toutes les valeurs possibles de c_1 et c_2.

Enfin, il faut définir $\mathcal{M}$, c'est à dire associer à chaque arcs et noeuds du CFG une fonction de $\mathcal{F}$. On va associer à un arc $\langle x, y \rangle$ entre deux blocs de base, la fonction $f_{\langle x,y \rangle}(t) = Used(y) \cup (t \cap NoTDef(y))$. L'algorithme que nous avons donné plus haut peux donc s'écrire :

for i=1 to $\#blocs$
    $Live(b_i) \leftarrow \emptyset$
$Changed \leftarrow True$
Tant que ($Changed$)
    $Changed \leftarrow False$
    for i=1 to $\#blocs$
        $OldValue \leftarrow Live(b_i)$
        $Live(b_i) = \bigwedge_{b_k \in succ(b_i)} f_{\langle b_i, b_k \rangle}(Live(b_k))$
        si $Live(b_i) \neq OldValue$ alors
            $Changed \leftarrow True$

On peut maintenant analyser plus formellement cet algorithme, c'est à dire montrer sa terminaison, sa correction et étudier son efficacité.

Terminaison Une chaîne décroissante est une séquence $x_1, x_3, \ldots, x_n$ ou $x_i \in \mathcal{L}$ et $x_i > x_{i+1}$. Considérons l'ensemble des chaînes décroissantes possibles dans $\mathcal{L}$, si la longueur des chaînes dans cet ensemble est bornée (par exemple par d), alors on dit que $\mathcal{L}$ possède la propriété des *chaînes décroissantes finies*. Ici, $\mathcal{L}$ étant fini, il a forcément la propriété des chaînes décroissantes finies.

On remarque ensuite que les fonctions dans $\mathcal{F}$ sont monotones : $x \leq y \Leftrightarrow x \wedge y = x, \Rightarrow f(x) = f(x \wedge y) = f(x) \wedge f(y)$ (en effet : $f(x \wedge y) = c_1 \cup ((x \cup y) \cap c_2) = c_1 \cup ((x \cap c_2) \cup (y \cap c_2)) = (c_1 \cup (x \cap c_2)) \cup (c_1 \cup (y \cap c_2)) = f(x) \wedge f(y)$). Et donc $x \leq y \Rightarrow f(x) = f(x) \wedge f(y) \Leftrightarrow f(x) \leq f(y)$ À

chaque étape de l'algorithme, on peut former un ensemble de chaînes décroissantes au sens large dans $\mathcal{L}$ en partant de $Live(b)$. Par exemple, à l'étape i :

$$Live_{etape_i}(b) = \bigwedge_{b_k \in succ(b)} f_{b,b_k}(Live_{etape_{i-1}}(b_k))$$

$$\Rightarrow \forall b_k \in succ(b), \quad Live_{etape_i}(b) \wedge f_{b,b_k}(Live_{etape_{i-1}}(b_k)) = Live_{etape_i}(b)$$
$$\Rightarrow \forall b_k \in succ(b), \;\; Live_{etape_i}(b) \leq f_{b,b_k}(Live_{etape_{i-1}}(b_k))$$

P uis :

$$\begin{aligned} \forall b_k \in succ(b_l) f_{b,b_k}(Live_{etape_{i-1}}(b_k)) &= f_{b,b_k}(\textstyle\bigwedge_{b_l \in succ(b_k)} f_{b_k,b_l}(Live_{etape_{i-2}}(b_l))) \\ &\leq \textstyle\bigwedge_{b_l \in succ(b_k)} f_{b,b_k}(f_{b_k,b_l}(Live_{etape_{i-2}}(b_l))) \end{aligned}$$

donc :

$$\forall b_k \in succ(b), \forall b_l \in succ(b_k), \;\; Live_{etape_i}(b) \leq f_{b,b_k}(Live_{etape_{i-1}}(b_k)) \leq f_{b_k,b_l}(f_{b_k,b_l}(Live_{etape_{i-2}}(b_l)))$$

etc.

Donc en déroulant l'algorithme on peut construire un ensemble de chaîne décroissantes terminant par les $Live(b_i)$ à l'étape courante, ces chaînes ont une longueur bornée donc l'algorithme termine forcement (les $Live(b_i)$ ne changent plus).

Correction Le concepteur doit ensuite donner du sens aux éléments de $\mathcal{L}$: si $v \in Live(b_i)$, alors il existe un chemin depuis la sortie du bloc b_i jusqu'à une opération qui utilise la valeur de v et suivant lequel v n'est pas redéfini (appelons cela un chemin v-libre). La solution donnée par l'algorithme doit être telle que pour chaque variable $v \in Live(b)$, il existe un chemin v-libre de b à une utilisation de v. La plupart des problèmes data-flow sont formulés en utilisant l'ensemble de tous les chemins possibles ayant une certaine propriété. On parle de *meet over all path solution*. Comment relier le point fixe donné par l'exécution de l'algorithme (qui assure une propriété entre voisin) à la propriété ci-dessus qui est une propriété sur un ensemble de chemin ?

La méthode habituelle est d'arriver à prouver que la solution que l'on recherche est un point fixe extrémal de l'équation. Dans notre exemple, le point fixe maximal correspondra à des ensembles avec le moins d'éléments possibles. Or on peut montrer (sous certaines hypothèses qui sont remplies ici) que l'algorithme que l'on propose va précisément converger vers ce point fixe extrémal.

En déroulant un nombre infini de fois l'équation (2), on trouve la propriété équivalente :

$$Live(b) = \bigwedge_{chemin\ p\ de\ n_f\ a\ b} f_p(\emptyset) \tag{3}$$

Ou n_f est le bloc de sortie du CFG et f_p est la composition des fonctions f induites par les arcs du chemin de n_f à p (le chemin est parcouru dans le sens opposé aux arcs : le problème est remontant). En effet, on doit avoir $Live(n_f) = \emptyset$ et on a l'équation (2) qui spécifie la relation entre $Live(b)$ et ses suivants.

On voit facilement que si les ensemble $Live(b)$ sont définis par l'équation (3) alors il forment un point fixe de l'équation (2) :

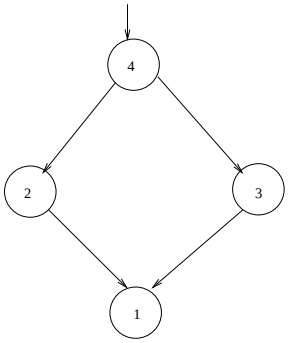
$$\begin{aligned} \textstyle\bigcup_{s \in succ(b)} Used(s) \cup (Live(s) \cap NotDef(s)) &= \textstyle\bigwedge_{s \in succ(b)} Used(s) \wedge (\bigwedge_{p:n_f \leftarrow s} f_p(\emptyset) \cap NotDef(s)) \\ &= \textstyle\bigwedge_{s \in succ(b)} f_{\langle b,s \rangle}(\bigwedge_{p:n_f \leftarrow s} f_p(\emptyset)) \\ &= \textstyle\bigwedge_{p:n_f \leftarrow s} f_p(\emptyset) \\ &= Live(b) \end{aligned}$$

Donc en résumé :

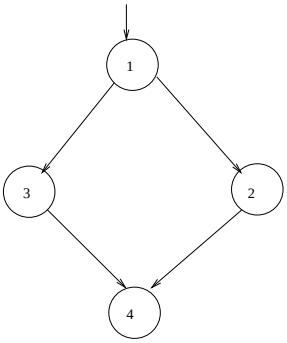
1. Comme le problème est admissible, l'algorithme de point fixe itératif donne l'unique plus grand point fixe du problème.
2. Comme la solution que nous recherchons est le plus grand point fixe du problème, c'est bien la solution donnée par l'algorihme

Complexité Le problème ayant un unique plus grand point fixe, la réponse ne dépend pas de l'ordre dans lequel l'algorithme calcule les ensembles $Live$ des blocs à chaque étape. Pour un problème remontant on aura intérêt à visiter les noeuds fils avant le noeud père (*postorder*). De même pour les ensemble $Avail$, on utilisera un ordre *inverse postorder* (profondeur d'abord).

Pour la plupart des analyses data flow, il y a deux ordres de parcours du graphe qui sont importants : les ordres *postorder* et *reverse postorder*. Un ordre postorder ordonne autant de des fils possibles d'un noeud (dans un certain ordre) avant de visiter le noeud, un ordre *reverse postorder* visite autant de pères possibles d'un noeud avant de visiter le noeud :



Postorder



Reverse postorder

Dans certain cas, on peut dire plus sur la complexité : si le problème a la propriété que

$$\forall f, g \in \mathcal{F}, \forall x \in \mathcal{F},\ \ f(g(\bot)) \geq g(\bot) \wedge f(x) \wedge (x)$$

alors l'algorithme s'arrêtera en au plus $d(G) + 3$ itérations ou $d(G)$ est le nombre maximal d'arc de retour dans un chemin acyclique du graphe. On parle de problème data flow *rapide* (*rapid framework*).

Notre problème est un problème rapide car si $f(x) = c_1 \cup (x \cap c_2)$, et $g(x) = c_3 \cup (x \cap c_4)$, $f(g(\bot)) = c_1 \cup ((c_3 \cup (\bot \cap c_4)) \cap c_2) = c_1 \cup ((c_3 \cup c_4)) \cap c_2)$ et d'autre part $g(\bot) \wedge f(x) \wedge (x) = (c_3 \cup (\bot \ \cap c_4)) \cup c_1 \cup (x \cap c_2) \cup x = c_3 \cup c_4 \cup c_1 \cup (x \cap c_2) \cup x = c_3 \cup c_4 \cup c_1 \cup x$ et on a bien $((c_3 \cup (\bot \cap c_4)) \cap c_2) \subset c_3 \cup c_4 \cup c_1 \cup x$ et donc $f(g(\bot)) \geq g(\bot) \wedge f(x) \wedge (x)$

Limitations L'analyse data flow suppose que tous les noeuds du CFG peuvent être pris, en pratique on peut souvent prouver à la compilation que de nombreux chemins ne sont jamais pris et donc améliorer le résultat obtenu par l'analyse data flow. On dit que l'information est précise *à hauteur* de l'exécution symbolique.

Un autre problème vient des imprécisions générées par les pointeurs et les tableaux. Une référence à un tableau $A[i, j, k]$ nomme un seul élémént : A. Sans une analyse poussée sur les valeurs de i,j et k le compilateur ne peut pas dire quel élément du tableau A est accédé. Dans ce cas, l'analyse doit être conservatrice.

Pour les pointeurs, si on ne sait pas ou pointent les pointeurs, une affectation à un objet pointé peut empêcher de conserver les variables dans des registres.

11.3 Autres problèmes data-flow

Reaching definitions Le compilateur a quelquefois besoin de connaître l'endroit où un opérande est défini. L'ensemble des définitions de cet opérande qui peuvent atteindre l'instruction que l'on étudie est appelé une *reaching definition*. Une définition d d'une variable v peut atteindre l'opération i si et seulement si i lit la valeur de v et qu'il existe un chemin v-libre entre d et i. L'équation data flow pour les ensemble $Reaches(n)$ qui contient l'ensemble des définitions pouvant atteindre le bloc n est :

$$Reaches(n) = \bigcup_{y \in preds(n)} Def(y) \cup (Reaches(y) \cap Survived(y))$$

Survived est conceptuellement la même chose que $NotKilled$ mais concerne des définitions plutôt que des expressions.

Very busy Une expressions e est considérée comme *très utilisée* (*very busy*) à un point p si e est évaluée et utilisée selon tous les chemins qui partent de p et que l'évaluation de e à p produirait le même résultat que l'évaluation e à l'endroit ou elle est évaluée (sur les différents chemins partant de p). L'analyse *very busy* est un problème remontant :

$$VeryBusy(b) = \bigcap_{a \in succ(b)} Used(a) \cup (VeryBusy(a) - Killed(a))$$

Propagation de constante La propagation de constante (*constant propagation*) peut avoir lieu quand le compilateur sait qu'une variable v atoujours la même valeur c à un point p du code. Le domaine de ce problème est un ensemble de paires : $\langle v_i, c_i \rangle$ ou v_i et une variable et c_i une constante ou une valeur spéciale $\perp$. L'analyse va annoter chaque noeud du CFG avec un ensemble $Constant(b)$ qui contient toutes les paires variable-constante que le compilateur peut prouver à l'entrée du bloc b. L'équation est la suivante :

$$Constant(b) = \bigwedge_{p \in preds(b)} f_p(Constant(p))$$

ou $\wedge$ compare deux paires :$\langle v_1, c_1 \rangle$ et $\langle v_1, c_2 \rangle$ de la manière suivante : $\langle v_1, \perp \rangle \wedge \langle v_1, c_2 \rangle = \langle v_1, \perp \rangle$ si c_1 ou c_2 vaut $\perp$ ou si $c_1 \neq c_2$ et $\langle v_1, c_1 \rangle \wedge \langle v_1, c_2 \rangle = \langle v_1, c_1 \rangle$ si $c_1 = c_2 \neq \perp$

La fonction f_p est un peu plus compliquée à spécifier car elle doit être détaillée en fonction de chaque instruction o_i du bloc, exemple :

$x \leftarrow y$ si $Constant(p) = \{\langle x, l_1 \rangle, \langle y, l_2 \rangle, \ldots\}$ alors $f_{o_i}(Constant(p)) = Constant(p) - \langle x, l_1 \rangle \cup \langle x, l_2 \rangle$

$x \leftarrow y \ op \ z$ si $Constant(p) = \{\langle x, l_1 \rangle, \langle y, l_2 \rangle, \langle z, l_3 \rangle, \ldots\}$ alors
$f_{o_i}(Constant(p)) = Constant(p) - \langle x, l_1 \rangle \cup \langle x, l_2 \ op \ l_3 \rangle$

Le compilateur peut utiliser le résultat de cette analyse pour évaluer certaines opérations à la compilation, éliminer des branchements lors de tests sur des valeurs constantes. C'est une des transformations qui peut amener des améliorations spectaculaires de performances.

12 Static Single Assignment

les analyses data flow prennent beaucoup de temps pour analyser les mêmes choses (utilisation de valeurs, etc.). La forme SSA permet de construire une représentation intermédiaire qui encode directement des informations sur le flot de contrôle *et* le flot de données.

Après passage en SSA, le code respecte deux règles : 1) chaque définition crée un nom unique et 2) chaque utilisation se réfère à une définition unique.

12.1 Passage en SSA

Un algorithme simple pour passer en SSA est le suivant :

1. insertion de fonctions ϕ : À chaque jointure du CFG, insérer des fonction ϕ pour tous les noms de variables. Par exemple au début d'un bloc ayant deux prédécesseurs, le compilateur insérera l'instruction $y \leftarrow \phi(y, y)$ pour chaque nom y. L'ordre dans lequel les fonctions sont introduites n'a pas d'importance car la sémantique dit que toutes les fonctions ϕ doivent lire leur argument en parallèle, puis écrire leurs résultats simultanément.
2. Renommage : Après avoir inséré les fonctions ϕ, le compilateur peut calculer les *reaching definition*, du fait que les fonctions ϕ sont au début des blocs, il y a nécessairement une définition par utilisation. On peut alors renommer les utilisations en fonction de la définition qui l'atteint (à l'intérieur d'un bloc), par exemple :

$$\begin{array}{c} y \leftarrow \phi(y,y) \\ \ldots \leftarrow y \end{array} \Rightarrow \begin{array}{c} y_1 \leftarrow \phi(y,y) \\ \ldots \leftarrow y_1 \end{array}$$

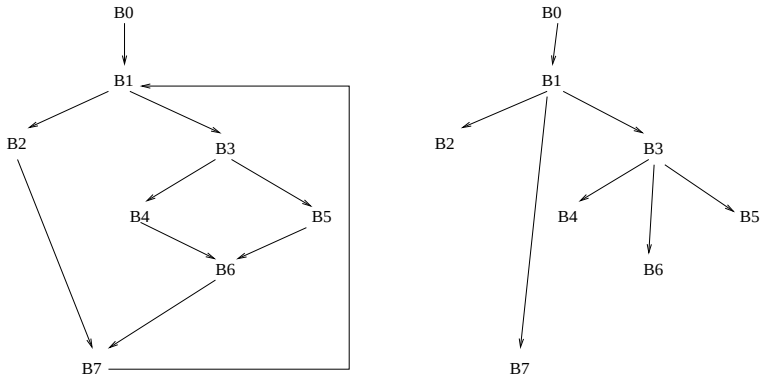
   Puis le compilateur trie les définitions de chaque variable v dans chaque bloc pour extraire celle qui atteint la fonction phi dans les successeurs du bloc.

La forme résultante est appelée la forme SSA maximale. Un des problème de la forme SSA maximale est qu'elle contient beaucoup trop de fonctions ϕ. On va maintenant voir comment on construit une SSA semi-simplifiée (*semi-pruned* SSA). La clé pour réduire le nombre de fonctions ϕ est d'utiliser la notion de *frontière de dominance*.

frontière de dominance On peut calculer les ensembles $Dom(b)$ des blocs qui dominent le bloc B par une équation data flow simple :

$$Dom(b) = \bigcap_{p \in preds(n)} (\{n\} \cup Dom(p))$$

avec les conditions initiales $Dom(n) = \{n\}$ pour tout n. Ce problème est un problème rapide descendant (très proche de $Live$). Par exemple, pour le graphe suivant, il trouve la solution en une itération (pour un order $postorder$ particulier du graphe) :



$Iteration/bloc$	0	1	2	3	4	5	6	7
–	$\{0\}$	$\{1\}$	$\{2\}$	$\{3\}$	$\{4\}$	$\{5\}$	$\{6\}$	$\{7\}$
1	$\{0\}$	$\{0,1\}$	$\{0,1,2\}$	$\{0,1,3\}$	$\{0,1,3,4\}$	$\{0,1,3,5\}$	$\{0,1,3,6\}$	$\{0,1,7\}$
2	$\{0\}$	$\{0,1\}$	$\{0,1,2\}$	$\{0,1,3\}$	$\{0,1,3,4\}$	$\{0,1,3,5\}$	$\{0,1,3,6\}$	$\{0,1,7\}$

On peut maintenant déterminer plus précisément quelle variable va nécessiter une fonction ϕ à chaque jointure du CFG. Considérons une définition d'une variable dans un bloc n du CFG, cette valeur peut atteindre toutes les instructions des noeuds m pour lesquels $n \in Dom(m)$ sans nécessiter une fonction ϕ puisque chaque chemin qui passe par m passe par n (si c'est une autre définition de la même variable, elle écrase la définition du bloc n).

Une définition dans un noeud n du CFG force une fonction ϕ dans un noeud qui rassemble des noeuds qui sont en dehors de la région du CFG que n domine. Plus formellement : une définition dans le noeud n force la création d'une fonction ϕ dans tout noeud m ou (1) n domine un prédécesseur de m et (2) n ne domine pas strictement $n \notin Dom(m) - \{m\}$. L'ensemble des noeuds ayant cette propriété par rapport à n est appelée la *frontière de dominance de* n (*dominance frontier*) $DF(n)$.

Pour calculer une frontière de dominance $DF(n)$, on considère les propriétés suivantes : Les noeuds de $DF(n)$ doivent être des noeuds de jointure (ayant plusieurs pères), les prédécesseurs d'un tel noeud $j \in DF(n)$ doivent avoir le noeud j dans leur frontière de dominance sinon ils domineraient j (et comme n les dominent, n dominerait j). Enfin, les dominateurs des prédecesseurs de j doivent aussi avoir j dans leur frontière de dominance car sinon il domineraient j aussi. On en déduit l'algorithme suivant :

1. Identifier les noeuds de jointure j (noeuds à plusieurs prédécesseurs).
2. Examiner chaque prédécesseur p de j dans le CFG. On remonte alors dans l'arbre de domination en rajoutant j dans la frontière de dominance du noeud courant tant que l'on ne rencontre pas le dominateur immédiat de j. (intuitivement, le reste des dominateurs de j sont partagés par tous les prédécesseurs de j, comme ils dominent j il n'auront pas j dans leur frontière de dominance).

L'algorithme est le suivant :

pour chaque noeud b
    Si le nombre de prédécesseurs de b est > 1
        pour chaque prédécesseurs p de b
            $courant \leftarrow p$
            Tant que $courant \neq iDom(b)$
                ajouter b à $DF(courant)$
                $courant \leftarrow iDom(courant)$

Placement des fonctions ϕ Considérons un ensemble de noeuds S contentant toutes les définitions de la variable x. Appelons $J(S)$ l'*ensemble des noeuds joints de* S (*set of joint nodes of* S) défini par $J(S)$={noeud z du CFG tel qu'il existe deux chemins p_{xz} et p_{yz} ayant z comme premier noeud commun (pour tout $x \in S$ et $y \in S$)}. Tous les noeud dans $J(S)$ vont nécessiter l'ajout d'une fonction ϕ pour la définition de x.

Mais cette nouvelle fonction ϕ crée une nouvelle définition de x, donc on doit de nouveau calculer un ensemble de noeud joint, on définit ainsi $J^+(S)$: l'*ensemble itéré des noeuds joints* de J comme la limite de la suite : $J_1 = J(S)$ et $J_{i+1} = J(S \cup J_i)$. $J^+(S)$ est exactement l'ensemble des noeuds nécessitant une fonction ϕ pour la définition de x. Un résultat important pour placer les ϕ est le suivant, on peut montrer que :

$$J^+(S) = DF^+(S)$$

ou $DF^+(S)$ est la *frontière de dominance itérée*, obtenu de la même manière comme limite de la suite : $DF_1 = DF(S)$ et $DF_{i+1} = DF(S \cup DF_i)$ (on voit mieux maintenant pourquoi on s'est intéresse au calcul efficace de la frontière de dominance des noeuds).

Renommage des variables Une fois que les fonctions ϕ sont placées, il faut renommer les variables pour passer en assignation unique (pour l'instant le code est plein d'instructions du type $x \leftarrow \phi(x, x)$). On fait cela par un parcours *pre-order* en maintenant une liste de nom pour chaque variable.

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).

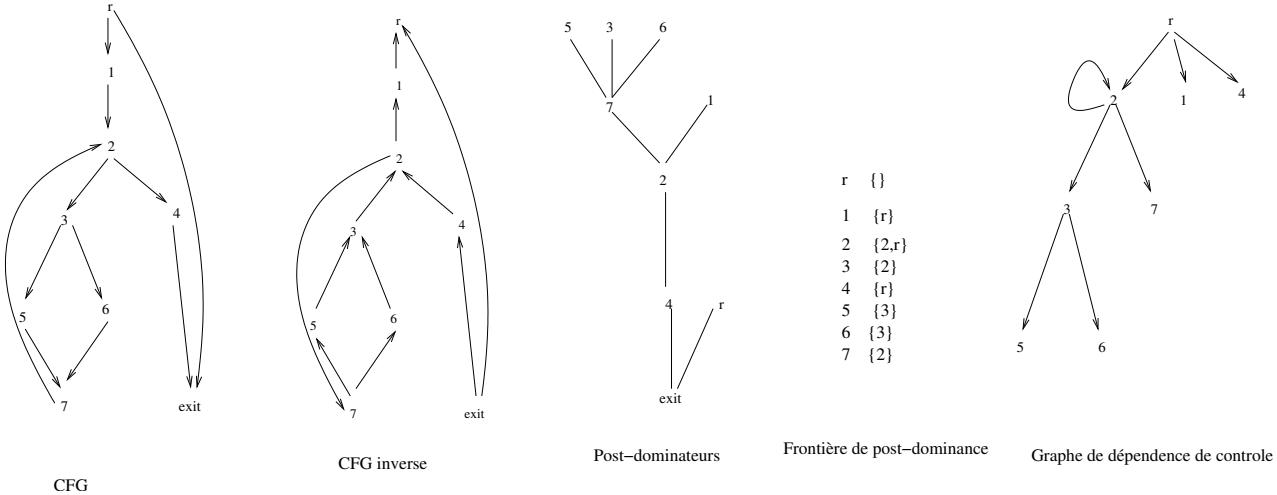
Élimination de code inutile L'élimination de code inutile se fait un peu comme la gestion du tas (*mark-sweep collector*), elle agit en deux passes. La première passe commence par repérer les opération *critiques* : entrées/sorties, appels de procédure, codes d'entrée et de sortie des procédures, passages 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 quite 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 dominace 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 :

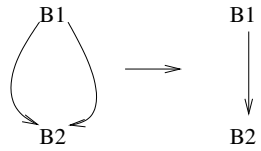


Lorsqu'une opération d'un bloc b_i est marquée comme utile par la première passe, on visite chaque bloc sur la frontière de dominance inversée de b_i. Chacun de ces blocs se finit par un

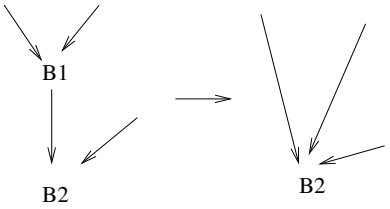
branchement qui est utile pour arriver au bloc b_i, donc on marque les branchements de fin de blocs comme utiles pour tous ces blocs et on les rajoute à la liste d'instructions à traiter.

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.

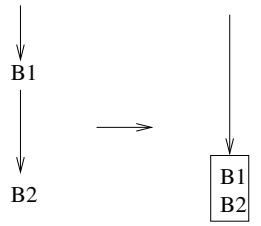
Élimination d'opérations de contrôle de flot inutile Cette opération fait appel à quatre transformations (voir schéma ci-dessous). Le court circuit (*branch hoisting*) arrive lorsque l'on rencontre un bloc vide qui contient un branchement en fin de bloc, on peut alors recopier le branchement dans les 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.



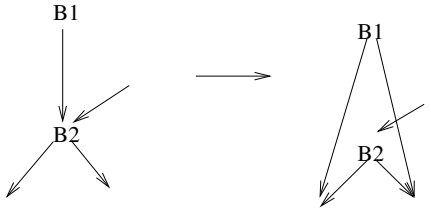
Branchement redondant



Suppression d'un bloc vide

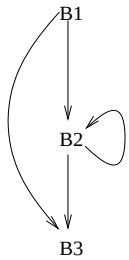


Agregation de blocs

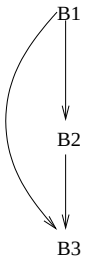


Court−circtuit d'un bloc vide

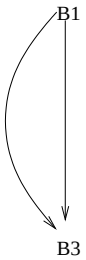
Aucune de ces transformations ne permet de supprimer une boucle vide : considérons par exemple le graphe ci-dessous et supposons que $B2$ soit vide. Le branchement de B2 n'est pas redondant, $B2$ ne termine pas par un saut, il ne peut donc pas être combiné avec $B3$. son prédécesseur $B1$ finit par un branchement, il ne peut pas être fusionné avec et il ne peut pas être court-circuité.



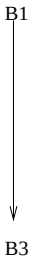
Original



code inutile



suppression B2



suppression redondance

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 :

$sum \leftarrow 0$
for $i \leftarrow 1$ to 100
    $sum \leftarrow sum + a(i)$

Un exemple code Iloc pouvant être généré après passage en SSA est donné ici (on suppose que le tableau est déclaré entre en $a[1..100]$) :

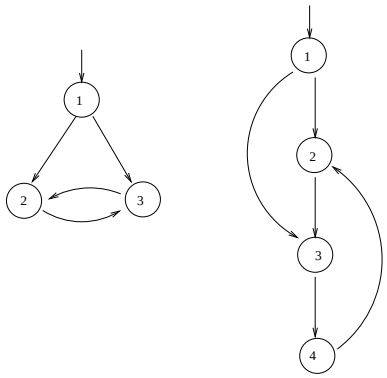
	$loadI$	0	$\Rightarrow$	r_{s0}		r_{s0}	$\leftarrow$	0
	$loadI$	1	$\Rightarrow$	r_{i0}		r_{i0}	$\leftarrow$	1
	$loadI$	100	$\Rightarrow$	r_{100}		r_{100}	$\leftarrow$	100
l_1	phi	r_{i0}, r_{i2}	$\Rightarrow$	r_{i1}	l_1	r_{i1}	$\leftarrow$	$phi(r_{i0}, r_{i2})$
	phi	r_{s0}, r_{s2}	$\Rightarrow$	r_{s1}		r_{s_1}	$\leftarrow$	$phi(r_{s0}, r_{s2})$
	$subI$	$r_{i1}, 1$	$\Rightarrow$	r_1		r_1	$\leftarrow$	$r_{i1} - 1$
	$multI$	$r_1, 4$	$\Rightarrow$	r_2		r_2	$\leftarrow$	$r_1 * 4$
	$addI$	$r_2, @a$	$\Rightarrow$	r_3		r_3	$\leftarrow$	$r_2 + @a$
	$load$	r_3	$\Rightarrow$	r_4		r_4	$\leftarrow$	$mem(r_3)$
	add	r_4, r_{s1}	$\Rightarrow$	r_{s2}		r_{s2}	$\leftarrow$	$r_4 + r_{s1}$
	$addI$	$r_{i1}, 1$	$\Rightarrow$	r_{i2}		r_{i2}	$\leftarrow$	$r_{i1} + 1$
	cmp_LE	r_{i2}, r_{100}	$\Rightarrow$	r_5		r_5	$\leftarrow$	$r_{i2} <_? r_{100}$
	cbr	r_5	$\rightarrow$	l_1, l_2		$cbr\ r_5$	$\rightarrow$	l_1, l_2
l_2	$\ldots$				l_2	$\ldots$		

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 : $@a, @a+4, @a+8, ..., @a+396$. 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 :

$$x \leftarrow i \times j \quad x \leftarrow j \times i \quad x \leftarrow i \pm j \quad x \leftarrow j + i$$

ou 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 region de la boucle alors la variable est une constante de région.

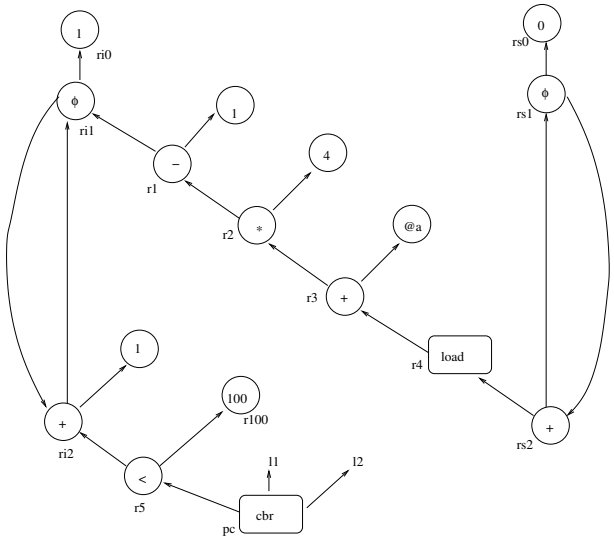
Pour déterminer si une variable est une variable d'induction, l'algorithme vérifie la forme des instructions qui mettent à jour la variable, les formes de mises à jours autorisées sont :

1. une variable d'induction plus une constante de région
2. une variable d'induction moins une constante de région
3. une fonction ϕ 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 utilisation appelé aussi graphe de SSA). Pour notre code, le graphe est le suivant :

$$
\begin{array}{lll}
 & r_{s0} & \leftarrow 0 \\
 & r_{i0} & \leftarrow 1 \\
 & r_{100} & \leftarrow 100 \\
l_1 & r_{i1} & \leftarrow phi(r_{i0}, r_{i2}) \\
 & r_{s_1} & \leftarrow phi(r_{s0}, r_{s2}) \\
 & r_1 & \leftarrow r_{i1} - 1 \\
 & r_2 & \leftarrow r_1 * 4 \\
 & r_3 & \leftarrow r_2 + @a \\
 & r_4 & \leftarrow mem(r_3) \\
 & r_{s2} & \leftarrow r_4 + r_{s1} \\
 & r_{i2} & \leftarrow r_{i1} + 1 \\
 & r_5 & \leftarrow r_{i2} <_? r_{100} \\
 & cbr\ r_5 & \rightarrow l_1, l_2 \\
l_2 & \ldots &
\end{array}
$$



Chaque variable d'induction est nécessairement définie dans une composante fortement connexe (CFC) du graphe de SSA, l'algorithme peut identifier toutes le 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 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 finit 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.

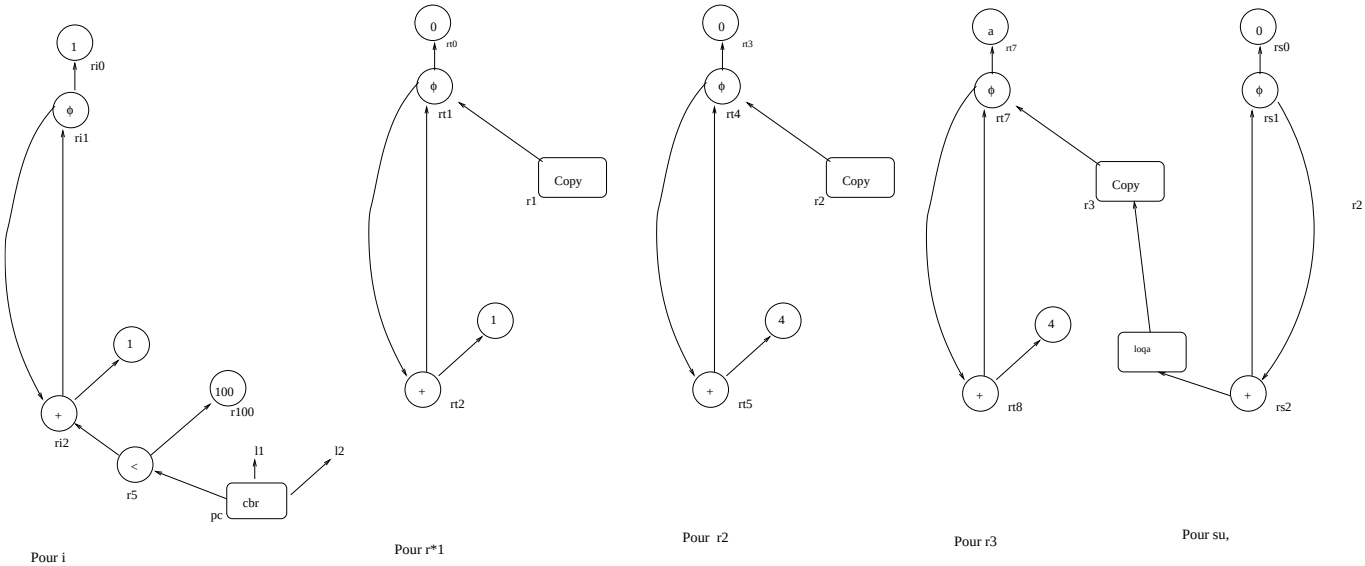
Cette méthode pour trouver les CFC a une propriété intéressante : lorsqu'une CFC est dépilée, DFS à déja visité tous ses enfants. Les opérations candidates sont donc visitées après que leurs arguments aient été traités. On va donc repérer les variables d'induction avant d'essayer de réduire les opérations 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 récursifs à DFS sur les noeuds r_4, r_3, r_2, r_1 et r_{i1}. À r_{i1} il fait des appels récursifs 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 $r_1 \leftarrow r_{t1}$ 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 :



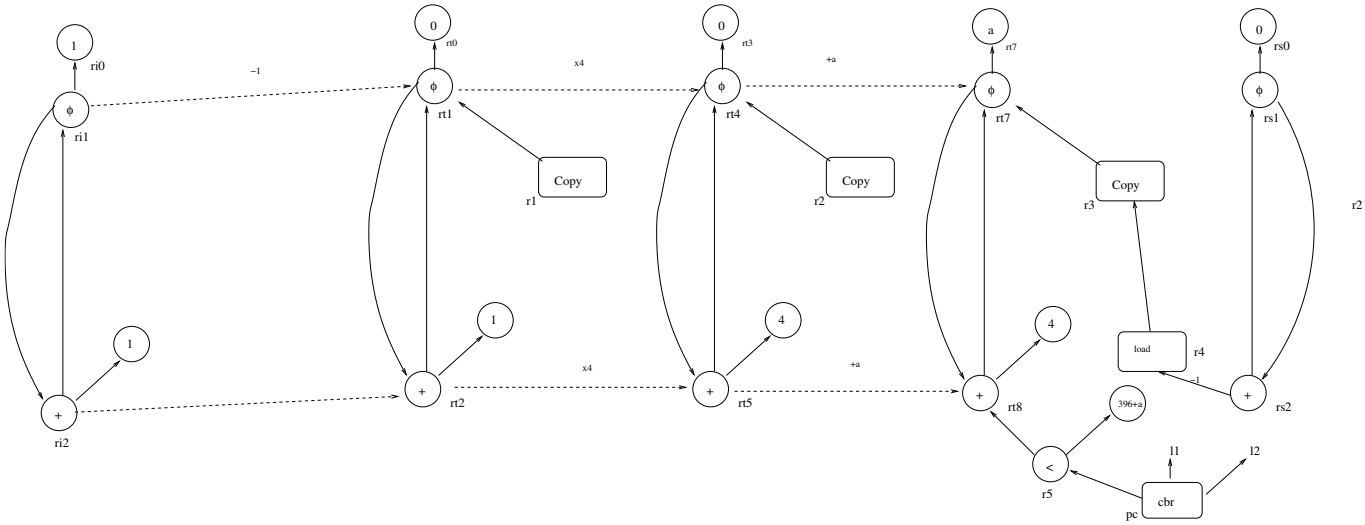
Ensuite, on essaye de réduire $r_2 \leftarrow r_1 \times 4$, cette opération est candidate, la procédure $Replace$ duplique la variable d'induction r_1, ajuste l'incrément en fonction de la multiplication et ajoute une copie vers r_2. L'instruction $r_3 \leftarrow r_2 + \partial a$ duplique l'induction 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) réécrire 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 :

	$loadI$	0	$\Rightarrow$	r_{s0}
	$loadI$	$@a$	$\Rightarrow$	r_{t6}
	$addI$	$r_{t6}, 396$	$\Rightarrow$	r_{lim}
l_1	phi	r_{t6}, r_{t8}	$\Rightarrow$	r_{t7}
	phi	r_{s0}, r_{s2}	$\Rightarrow$	r_{s_1}
	$load$	r_{t7}	$\Rightarrow$	r_4
	add	r_4, r_{s1}	$\Rightarrow$	r_{s2}
	$addI$	$r_{t7}, 4$	$\Rightarrow$	r_{t8}
	cmp_LE	r_{t7}, r_{lim}	$\Rightarrow$	r_5
	cbr	r_5	$\rightarrow$	l_1, l_2
l_2	$\ldots$			

14 Ordonnancement d'instructions

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	$\Rightarrow r_1$	// load w
4	add	r_1, r_1	$\Rightarrow r_1$	// $r_1 \leftarrow w \times 2$
5	loadAI	r_{arp}, @x	$\Rightarrow r_2$	// load x
8	mult	r_1, r_2	$\Rightarrow r_1$	// $r_1 \leftarrow (w \times 2) \times x$
9	loadAI	r_{arp}, @y	$\Rightarrow r_2$	// load y
12	mult	r_1, r_2	$\Rightarrow r_1$	// $r_w \leftarrow (w \times 2) \times x \times y$
13	loadAI	r_{arp}, @z	$\Rightarrow r_2$	// load z
16	mult	r_1, r_2	$\Rightarrow r_1$	// $r_1 \leftarrow (w \times 2) \times x \times y \times z$
18	storeAI	r_1	$\Rightarrow r_{arp}$, @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	$\Rightarrow r_1$	// load w
2	loadAI	r_{arp}, @x	$\Rightarrow r_2$	// load x
3	loadAI	r_{arp}, @y	$\Rightarrow r_3$	// load y
4	add	r_1, r_1	$\Rightarrow r_1$	// $r_1 \leftarrow w \times 2$
5	mult	r_1, r_2	$\Rightarrow r_1$	// $r_1 \leftarrow (w \times 2) \times x$
6	loadAI	r_{arp}, @z	$\Rightarrow r_2$	// load z
7	mult	r_1, r_3	$\Rightarrow r_1$	// $r_w \leftarrow (w \times 2) \times x \times y$
9	mult	r_1, r_2	$\Rightarrow r_1$	// $r_1 \leftarrow (w \times 2) \times x \times y \times z$
11	storeAI	r_1	$\Rightarrow r_{arp}$, @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 parallelisme 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.

L'ordonnanceur travaille sur le graphe de précédence $\mathcal{P} = (N, E)$ 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 donne un graphe de précédence, 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 :

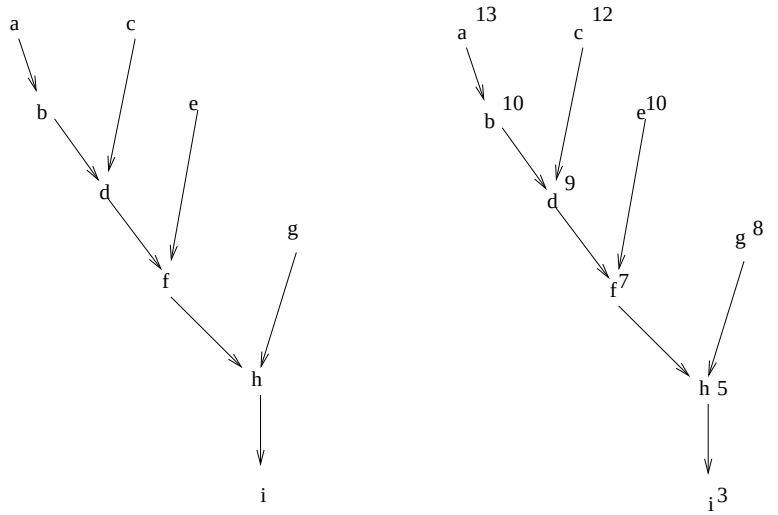
1. $S(n) \geq 0$ (on suppose aussi qu'il y a au moins une opération telle que $S(n) = 0$).
2. si $(n_1, n_2) \in E,\ S(n_1) + delai(n_1) \leq S(n_2)$. Cette contrainte assure la validité de l'ordonnancement.
3. L'ensemble des opérations s'exécutant à un cycle donné ne contient pas plus d'opérations de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

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

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

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 a 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'ordonnanceur a intérêt à ordonnancer l'instruction a en premier. Le chemin critique devient alors $cdefhi$ qui suggère d'ordonnancer 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 a l'ordonnancement $acebdgfhi$ qui est celui que l'on cherchait. Cependant, le compilateur ne peut pas réécrire le code avec cette ordonnancement en utilisant exactement les mêmes instructions. En effet, c et e définissent toutes les deux $r2$, 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-dependance* : 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'ordonnancement peut prendre cela en compte de deux manières : soit il respecte les anti-dependences comme les dépendances du graphe de précédence et n'ordonnance 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'ordonnancement influe sur la durée de vie des variables, Il doit à la fois ne pas ordonnancer une variable avant que ses opérandes ne soient prêts pour ne pas introduire de cycles inutilisés (*idle*) et ne pas ordonnancer 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*.

14.1 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 parcours 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 à 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 a 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 :

$Cycle \leftarrow 0$
$Ready \leftarrow$ feuilles de $\mathcal{P}$
$Active \leftarrow \emptyset$
Tant que $(Ready \cup Active \neq \emptyset)$
    si $Ready \neq \emptyset$ alors
        choisir une opération op dans $Ready$
        $S(op) \leftarrow Cycle$
        $Active \leftarrow Active \cup op$
    $Cycle \leftarrow Cycle + 1$
    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

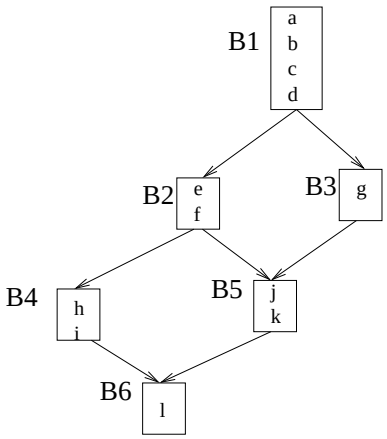
manière précise les égalités (lorsque plusieurs opérations ont la même priorité). Un bon algorithme de list scheduling aura plusieurs sortes de priorité pour résoudre les égalités. Il n'y a pas de priorité universelle, le choix dépend énormément du contexte et est difficile à prévoir.

- 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.

14.2 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'ordonnancement des blocs de base étendus est en fait un cas dégénéré du trace scheduling.

14.3 Ordonnancement de boucles

L'ordonnancement 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'ordonnancer 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^{ieme}, avec un décalage de $(199-1) \times 4 = 792$.

-4		$loadI$	$@x$	$\Rightarrow$	$r_{@x}$	
-3		$loadI$	$@y$	$\Rightarrow$	$r_{@y}$	// adresses et valeurs
-2		$loadI$	$@z$	$\Rightarrow$	$r_{@z}$	// préchargées
-1		$addI$	$r_{@x}, 792$	$\Rightarrow$	r_{ub}	
1	L_1 :	$loadAO$	$r_{sp}, r_{@x}$	$\Rightarrow$	r_x	// valeurs de $x[i]$ et $y[i]$
2		$loadAO$	$r_{sp}, r_{@y}$	$\Rightarrow$	r_y	
3		$addI$	$r_{@x}, 4$	$\Rightarrow$	$r_{@x}$	// incrément sur les adresses
4		$addI$	$r_{@y}, 4$	$\Rightarrow$	$r_{@y}$	// cachant la latence des chargements
5		$mult$	r_x, r_y	$\Rightarrow$	r_z	// le vrai travail
6		cmp_LT	$r_{@x}, r_{ub}$	$\Rightarrow$	r_{cc}	// test pendant la latence de la multiplication
7		$storeAO$	r_z	$\Rightarrow$	$r_{sp}, r_{@z}$	
8		$addI$	$r_{@z}, 4$	$\Rightarrow$	$r_{@z}$	
9		cbr	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		$loadI$	$@x$	$\Rightarrow$	$r_{@x}$	$loadI$	$@y$	$\Rightarrow$	$r_{@y}$
-1		$loadI$	$@z$	$\Rightarrow$	$r_{@z}$	$addI$	$r_{@x}, 792$	$\Rightarrow$	r_{ub}
1	L_1 :	$loadAO$	$r_{sp}, r_{@x}$	$\Rightarrow$	r_x	$addI$	$r_{@x}, 4$	$\Rightarrow$	$r_{@x}$
2		$loadAO$	$r_{sp}, r_{@y}$	$\Rightarrow$	r_y	$addI$	$r_{@y}, 4$	$\Rightarrow$	$r_{@y}$
3		nop				cmp_LT	$r_{@x}, r_{ub}$	$\Rightarrow$	r_{cc}
4		$mult$	r_x, r_y	$\Rightarrow$	r_z	$addI$	$r_{@z}, 4$	$\Rightarrow$	$r_{@z}$
5		nop				nop			
6		$storeAO$	r_z	$\Rightarrow$	$r_{sp}, r_{@z}$	nop			
7		cbr	r_{cc}	$\rightarrow$	L_1, L_2	nop			

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		$loadI$	$@x$	$\Rightarrow$	$r_{@x}$	$loadI$	$@y$	$\Rightarrow$	$r_{@y}$	// prologue
-3		$loadAO$	$r_{sp}, r_{@x}$	$\Rightarrow$	$r_{x'}$	$addI$	$r_{@x}, 4$	$\Rightarrow$	$r_{@x}$	// première itération (instruction 1)
-2		$loadAO$	$r_{sp}, r_{@y}$	$\Rightarrow$	r_y	$addI$	$r_{@y}, 4$	$\Rightarrow$	$r_{@y}$	// première itération (instruction 2)
-1		$loadI$	$@z$	$\Rightarrow$	$r_{@z}$	$addI$	$r_{@x}, 788$	$\Rightarrow$	r_{ub}	
0		nop				nop				
1	$L_1:$	$loadAO$	$r_{sp}, r_{@x}$	$\Rightarrow$	r_x	$addI$	$r_{@x}, 4$	$\Rightarrow$	$r_{@x}$	// valeur $x[i+1]$ et MAJ $@x = i+2$
2		$loadAO$	$r_{sp}, r_{@y}$	$\Rightarrow$	r_y	$mult$	$r_{x'}, r_y$	$\Rightarrow$	r_z	// valeur $y[i+1]$, calcul $x[i] * y[i]$
3		cmp_LT	$r_{@x}, r_{ub}$	$\Rightarrow$	r_{cc}	$addI$	$r_{@y}, 4$	$\Rightarrow$	$r_{@y}$	MAJ $@y = i+2$, copie insérée
4		$storeA0$	r_z	$\Rightarrow$	$r_{sp}, r_{@z}$	$i2i$	r_x	$\Rightarrow$	$r_{x'}$	// stocke $z[i]$, test iter. $i+1$
5		cbr	r_{cc}	$\rightarrow$	L_1, L_2	$addI$	$r_{@z}, 4$	$\Rightarrow$	$r_{@z}$	branchement itér. i, MAJ $@z = i+1$
$+1$		$mult$	$r_{x'}, r_y$	$\Rightarrow$	r_z	nop				// épilogue :fin dernière
$+2$		$storeA0$	r_z	$\Rightarrow$	$r_{sp}, r_{@z}$	nop				// 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 :

$cycle$	iteration i		iteration i+1		iteration i+2	
1			$r_x \leftarrow [@x]$	$@x \leftarrow i+2$		
2		$r_z \leftarrow r_{x'} * r_y$	$r_y \leftarrow [@y]$			
3	$cmp\_LT\ r_{ub}$			$@y \leftarrow i+2$		
4	$[@z] \leftarrow r_z$			$r_{x'} \leftarrow r_x$		
5	$cbr\ \ L_1, L_2$	$@z \leftarrow i+1$				
6					$r_x \leftarrow [@x]$	$@x \leftarrow i+3$
7				$r_z \leftarrow r_{x'} * r_y$	$r_y \leftarrow [@y]$	
8			$cmp\_LT\ r_{ub}$			$@y \leftarrow i+3$
9			$[@z] \leftarrow r_z$			$r'_x \leftarrow r_x$
10			$cbr\ \ L_1, L_2$	$@z \leftarrow i+2$		
11						
12						$r_z \leftarrow r_{x'} * r_y$
13					$cmp\_LT\ r_{ub}$	
14					$[@z] \leftarrow r_z$	
15					$cbr\ \ L_1, L_2$	$@z \leftarrow i+3$
16						

On voit que les opérations de deux itérations peuvent se recouvrir facilement, ci dessous, en gras les opérations de l'étape $i+1$:

6	$\mathbf{r_x \leftarrow [@x]}$	$\mathbf{@x \leftarrow i+3}$
7	$\mathbf{r_z \leftarrow r_{x'} * r_y}$	$r_y \leftarrow [@y]$
8	$cmp\_LT\ r_{ub}$	$\mathbf{@y \leftarrow i+3}$
9	$r_z \leftarrow [@z]$	$\mathbf{r'_x \leftarrow r_x}$
10	$cbr\ \ L_1, L_2$	$@z \leftarrow i+1$

L'implémentation du pipeline logiciel a fait l'objet de nombreuses recherches et continu à être un domaine très actif.

15 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 et produit un programme équivalent qui est compatible avec l'ensemble de registres disponibles sur la machine cible. 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 est appelé *spill*. Les valeurs ambiguës (pouvant être pointées par plusieurs variables) vont générer des spill obligatoirement, la surpression de valeurs ambiguës peut améliorer significativement les performances du code généré.

Le traitement de l'allocateur recouvre en fait deux traitements distincts : l'allocation de registres et l'assignation de registres. Ces deux traitements sont généralement faits en une seule passe.

L'allocation de registres envoi un nombre non limité de noms de registres sur un nombre fini de noms virtuels correspondant au nombre de ressources disponibles sur la machine cible. 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.

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

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*). Beaucoup d'architectures imposent que les paramètres des fonctions soient passés dans certains registres, les registres de code conditions sont aussi spécifiques. L'IA-64 a une classe de registres spécifiques pour la prédication et pour les adresses cibles de branchement.

15.1 Allocation de registres locale

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 $op_1, \ldots, op_N$ 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 designe 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 retourner un nom de registre physique utilisé pour chaque registre virtuel rencontré, 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
        $result \leftarrow allocate(vr, class)$
        emettre le code pour charger vr dans $result$
    return $result$

Si l'on oubli 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-t'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];
    int Next[Size];
    int Free[Size];
    int Stack[Size];
    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 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 :

$$\begin{array}{l} store\ x_2 \\ load\ x_3 \\ load\ x_2 \end{array}$$

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

$$\begin{array}{l} load\ x_3 \\ load\ x_1 \end{array}$$

Ce scenario suggère de spiller les valeurs propres d'abord, mais on peut trouver des cas ou 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 :

$$\begin{array}{l} load\ x_3 \\ load\ x_1 \\ load\ x_3 \\ load\ x_1 \end{array}$$

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

$$\begin{array}{l} store\ x_2 \\ load\ x_3 \\ load\ x_2 \end{array}$$

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 $allocate$

Notion de durée de vie Lorsque l'on prend en compte plusieurs blocs, on ne peut plus faire l'hypothèse que les données ne sont plus utilisées après la fin du bloc, il faut donc utiliser une notion plus compliquée : la notion de durée de vie (*live range*, aussi appelée *web*) basée sur la vivacité (*liveness*).

une *durée de vie* est constituée d'un ensemble de définitions et d'utilisations d'une valeur qui sont reliés dans le sens que si une définition peut atteindre une utilisation, elle sont dans la même durée de vie. On rappelle la définition de vivacité :

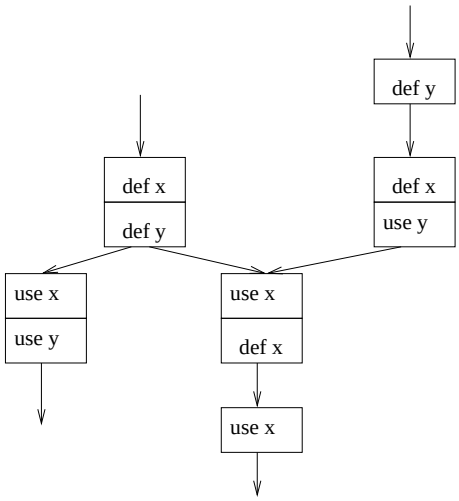
À un point p d'une procédure, une valeur est vivante si elle a été définie sur un chemin allant du début de la procédure à p et il existe un chemin de p à une utilisation de v selon lequel v n'est pas redéfini.

Considérons le code suivant :

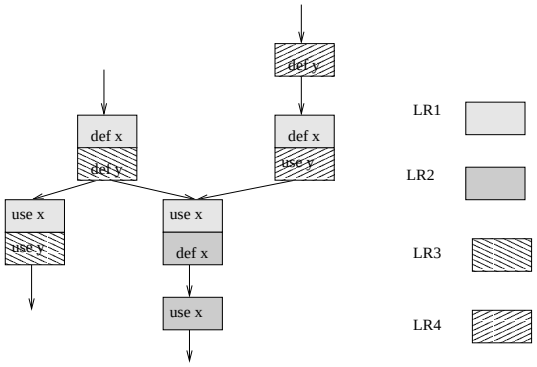
1.	$loadI$	$@stack$	$\Rightarrow$	r_{arp}
2.	$loadAI$	$r_{arp}, 0$	$\Rightarrow$	r_w
3.	$loadI$	2	$\Rightarrow$	r_2
4.	$loadAI$	$r_{arp}, 8$	$\Rightarrow$	r_x
5.	$loadAI$	$r_{arp}, 16$	$\Rightarrow$	r_y
6.	$loadAI$	$r_{arp}, 24$	$\Rightarrow$	r_z
7.	$mult$	r_w, r_2	$\Rightarrow$	r_w
8.	$mult$	r_w, r_x	$\Rightarrow$	r_w
9.	$mult$	r_w, r_y	$\Rightarrow$	r_w
10.	$mult$	r_w, r_z	$\Rightarrow$	r_w
11.	$storeAI$	r_w	$\Rightarrow$	$r_{arp}, 0$

La valeur de r_{arp} est définie une fois au début du bloc, ensuite elle est utilisée jusqu'à la fin du bloc. On en déduit la durée de vie $[1, 11]$. Considérons r_w, 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 r_2, $[4, 8]$ pour r_x, $[5, 9]$ pour r_y, $[6, 10]$ pour r_z.

Lorsque l'on considère plusieurs blocs de base, les durées de vie ne sont plus des intervalles. Considérons l'exemple suivant :



Les durées de vie sont les suivantes :



Chaque valeur du programme a une durée de vie, même si elle n'a pas de nom dans le programme (par exemple les valeurs intermédiaires produites par les calculs d'adresse). Une variable peut avoir comme durée de vie plusieurs intervalles distincts correspondant chacun à la durée de vie d'une valeur différente stockée dans la variable, on peut alors renommer la variable en autant de noms différents qu'il y a de durées de vie. On peut donc identifier les valeurs à stocker dans les registres (variables, valeurs, temporaires, constantes, etc.) et les durées de vie.

L'allocateur de registre va utiliser ces durées de vie pour placer cette variable dans differents registres.

On peut facilement trouver les valeurs tuées ou définies dans chaque bloc. Pour un ensemble de bloc, le compilateur doit découvrir les valeurs vivantes en entrée de chaque bloc et les valeurs vivantes en sortie de chaque bloc : $LiveIn(b)$ et $LiveOut(b)$ en utilisant une analyse data flow.

L'allocateur travaillant sur une région peut alors utiliser ces informations : une valeur vivante en fin d'un bloc doit être stockée en mémoire. Cela peut amener des stockages suivis de chargements redondants, mais cela est assez difficile à détecter sans avoir une vision globale.

15.2 Allocation globale

L'évaluation d'une allocation de registre est délicate car elle peut se révéler plus ou moins bonne suivant les exécutions. L'allocateur global diffère de l'allocateur local en deux points importants : Le calcul des durées de vie est plus complexe (les durées de vie ne sont plus que des intervalles) et le coût d'un spill n'est plus uniforme car plusieurs références à une même variable peuvent ne pas s'exécuter le même nombre de fois.

Pour découvrir les durée de vie, il semble judicieux d'utiliser la SSA. les fonctions ϕ rendent explicite le fait que des définitions distinctes sur des chemins distincts atteignent une même référence. Deux définitions atteignant la même fonction ϕ 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 assigne un ensemble diffèrent à chaque nom. Ensuite, il examine les fonctions ϕ du programme et fait l'union des ensembles pour tous les paramètres d'une même fonction ϕ. Lorsque toutes les fonctions ϕ ont été traitées, les ensembles résultant représentent les durées de vie maximales du programme.

Le compilateur doit avoir une évaluation du coût d'un spill. En général les données sont allouées sur la pile, ce qui évite d'utiliser d'autres registres pour le calcul de l'adresse. Le compilateur peut alors évaluer pour une durée de vie particulière, le coût du spill d'une 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, une utilisation qui suit immédiatement une définition, aucun spill ne pourra réduire le nombre de registres utilisé. 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 ou k durées de vie qui n'interfèrent pas ont toutes un coût infini).

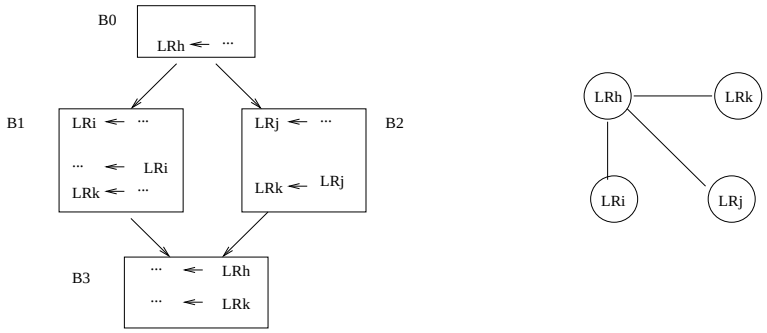
Enfin, pour tenir compte de la fréquence d'utilisation des différents blocs la plupart des compilateurs utilisent des heuristiques assez primaires : chaque boucle s'exécute 10 fois (donc une double boucle 100 fois), les branchements sont pris une fois sur deux, etc. Le coût du spill d'une

référence est alors :

coût des opérations mémoire ajoutées $\times$ *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 ou 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 noeuds 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 les variables correspondantes sont toute deux vivantes à une même opération. Considérons l'exemple suivant :



Si le compilateur peut construire une k-coloration du graphe d'interférence, c'est à dire colorer chaque noeud 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 noeud. Il peut soit spiller la variable soit *séparer* la durée de vie. Splitter 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 Une première approche consiste à partitionner le graphe en deux types de noeuds : les noeuds non contraints qui ont moins de $k-1$ voisins et les noeuds contraints qui ont au moins k voisins. Quelque soient les couleurs des noeuds contraints, les noeuds non contraints pourront toujours être colorés. Les premiers noeuds contraints à être colorés doivent être ceux dont les temps estimés de spill sont les plus importants. Cet allocateur alloue les couleurs suivant cet ordre de priorité. Lorsqu'il rencontre un noeud qu'il ne peut pas colorer, il sépare ou spill le noeud (pas de backtrack), en espérant que ce noeud sera plus facile à spiller que les noeuds déjà alloués. Séparer le noeud 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 $deg(n) < k$
$noeud \leftarrow n$
sinon
$noeud \leftarrow n$ choisi dans N
enlever $noeud$ et ses arcs de I
empiler $noeud$ sur la pile.

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

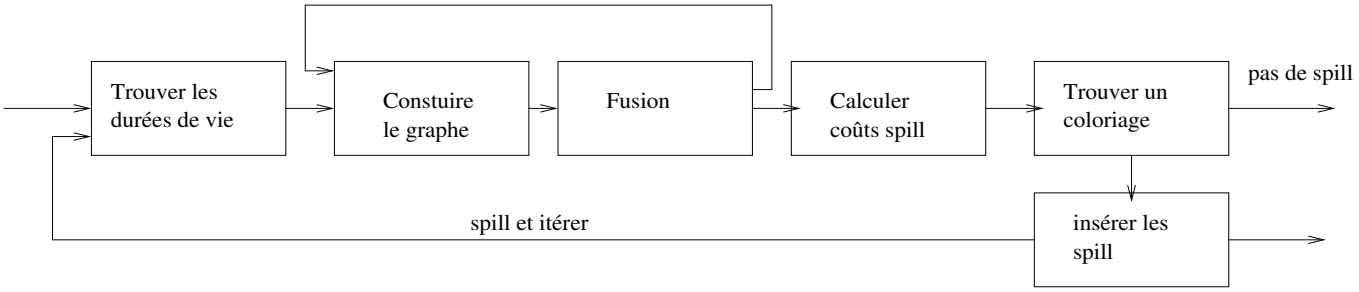
Tant que ($pile \neq \emptyset$)
$noeud \leftarrow sommet\ de\ pile$
insérer $noeud$ et ses arcs dans I
colorer le noeud

Si un noeud ne peut pas être coloré, il est laissé incolore. Lorsque le graphe a été reconstruit, si tous les noeuds sont colorés, l'allocation est terminée. Si il reste des noeuds 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 noeuds dans un ordre "intelligent". Pour cela, lorsque tous les noeuds du graphe sont contraints, il utilise la *spill metric* : il prend le noeud qui minimise le rapport *coût estimé du noeud/degré courant du noeud*.

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 noeuds 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 noeud 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.

Références

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

Opcode	*Sources*	*cible*	*Semantique*
nop	$--$	$--$	ne fait rien
add	r_1, r_2	r_3	$r_1 + r_2 \rightarrow r_3$
$addI$	r_1, c_1	r_2	$r_1 + c_1 \rightarrow r_2$
sub	r_1, r_2	r_3	$r_1 - r_2 \rightarrow r_3$
$subI$	r_1, c_1	r_2	$r_1 - c_1 \rightarrow r_2$
$mult$	r_1, r_2	r_3	$r_1 \times r_2 \rightarrow r_3$
$multI$	r_1, c_1	r_2	$r_1 \times c_1 \rightarrow r_2$
div	r_1, r_2	r_3	$r_1 \div r_2 \rightarrow r_3$
$divI$	r_1, c_1	r_2	$r_1 \div c_1 \rightarrow r_2$
$lshift$	r_1, r_2	r_3	$r_1 \ll r_2 \rightarrow r_3$
$lsiftI$	r_1, c_1	r_2	$r_1 \ll c_1 \rightarrow r_2$
$rshift$	r_1, r_2	r_3	$r_1 \gg r_2 \rightarrow r_3$
$rshiftI$	r_1, c_1	r_2	$r_1 \gg c_1 \rightarrow r_2$
and	r_1, r_2	r_3	$r_1 \wedge r_2 \rightarrow r_3$
$andI$	r_1, c_1	r_2	$r_1 \wedge c_1 \rightarrow r_2$
or	r_1, r_2	r_3	$r_1 \vee r_2 \rightarrow r_3$
orI	r_1, c_1	r_2	$r_1 \vee c_1 \rightarrow r_2$
xor	r_1, r_2	r_3	$r_1 xor\ r_2 \rightarrow r_3$
$xorI$	r_1, c_1	r_2	$r_1 xor\ c_1 \rightarrow r_2$
$loadI$	c_1	r_2	$c_1 \rightarrow r_2$
$load$	r_1	r_2	$Memory(r_1) \rightarrow r_2$
$loadAI$	r_1, c_1	r_2	$Memory(r_1 + c_1) \rightarrow r_2$
$loadAO$	r_1, r_2	r_3	$Memory(r_1 + r_2) \rightarrow r_3$
$cload$	r_1	r_2	$load$ pour caractères
$cloadAI$	r_1, c_1	r_2	$loadAI$ pour caractères
$cloadAO$	r_1, r_2	r_3	$loadAO$ pour caractères
$store$	r_1	r_2	$r_1 \rightarrow Memory(r_2)$
$storeAI$	r_1	r_2, c_1	$r_1 \rightarrow Memory(r_2 + c_1)$
$storeAO$	r_1	r_2, r_3	$r_1 \rightarrow Memory(r_2 + r_3)$
$cstore$	r_1	r_2	$store$ pour caractères
$cstoreAI$	r_1	r_2, c_1	$storeAI$ pour caractères
$cstoreAO$	r_1	r_2, r_3	$storeAO$ pour caractères
$i2i$	r_1	r_2	$r_1 \rightarrow r_2$
$c2c$	r_1	r_2	$r_1 \rightarrow r_2$
$c2i$	r_1	r_2	convertit un caractère en entier
$i2c$	r_1	r_2	convertit un entier en caractère

Pour le contrôle de flot :

Opcode	*Sources*	*cible*	*Semantique*
cbr	r_1	l_1, l_2	$r_1 = True \Rightarrow l_1 \rightarrow PC, \quad r_1 = False \Rightarrow l_2 \rightarrow PC$
jumpI	$--$	l_1	$l_1 \rightarrow PC$
jump	$--$	r_1	$r_1 \rightarrow PC$
cmp_LT	r_1, r_2	r_3	$r_1 < r_2 \Rightarrow True \rightarrow r_3, \quad r_1 \geq r_2 \Rightarrow False \rightarrow r_3$
cmp_LE	r_1, r_2	r_3	$r_1 \leq r_2 \Rightarrow True \rightarrow r_3$
cmp_EQ	r_1, r_2	r_3	$r_1 = r_2 \Rightarrow True \rightarrow r_3$
cmp_NE	r_1, r_2	r_3	$r_1 \neq r_2 \Rightarrow True \rightarrow r_3$
cmp_GE	r_1, r_2	r_3	$r_1 \geq r_2 \Rightarrow True \rightarrow r_3$
cmp_GT	r_1, r_2	r_3	$r_1 > r_2 \Rightarrow True \rightarrow r_3$
tbl	r_1, l_2	$--$	r_1 peut contenir l_1

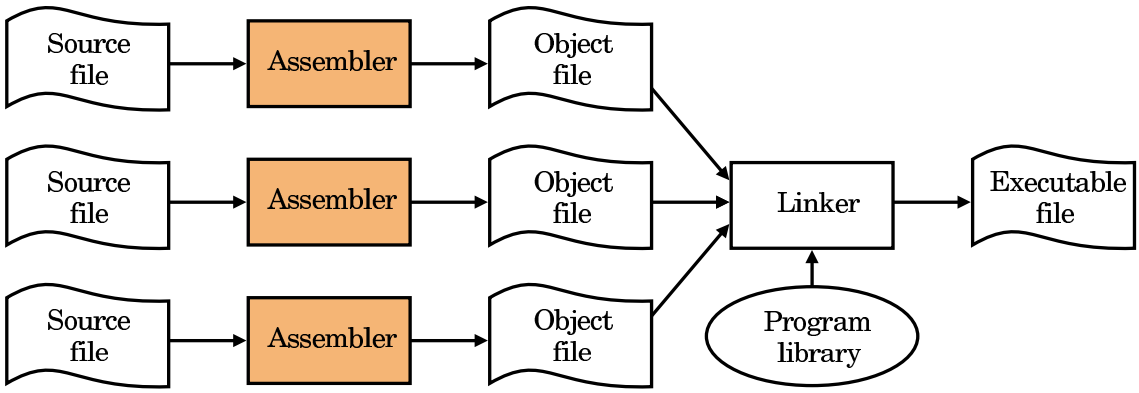
Autre syntaxe pour les branchements

Opcode	*Sources*	*cible*	*Semantique*
comp	r_1, r_2	cc_1	definit cc_1
cbr_LT	cc_1	l_1, l_2	$cc_1 = LT \Rightarrow l_1 \rightarrow PC, \quad cc_1 \neq LT \Rightarrow l_2 \rightarrow PC,$
cbr_LE	cc_1	l_1, l_2	$cc_1 = LE \Rightarrow l_1 \rightarrow PC$
cbr_EQ	cc_1	l_1, l_2	$cc_1 = EQ \Rightarrow l_1 \rightarrow PC$
cbr_GE	cc_1	l_1, l_2	$cc_1 = GE \Rightarrow l_1 \rightarrow PC$
cbr_GT	cc_1	l_1, l_2	$cc_1 = GT \Rightarrow l_1 \rightarrow PC$
cbr_NE	cc_1	l_1, l_2	$cc_1 = NE \Rightarrow l_1 \rightarrow PC$

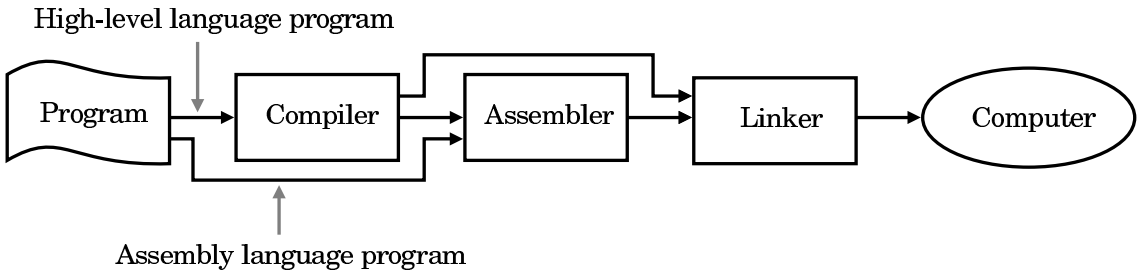
B Assembleur, éditeur de liens, l'exemple du Mips

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, *memoy 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`). 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*[2]est une représentation symbolique du langage machine (binaire) de la machine. Il permet en particulier l'utilisation d'étiquette (*labels*[3]) 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).

[2]Attention, en francais on utilise le même terme "assembleur" pour désigner le langage (assembleur ou langage d'assemblage) et l'outils qui compile le langage (assembleur)

[3]j'utiliserai le terme *labels* dans la suite

```
#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);
}
```

```
    .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)
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

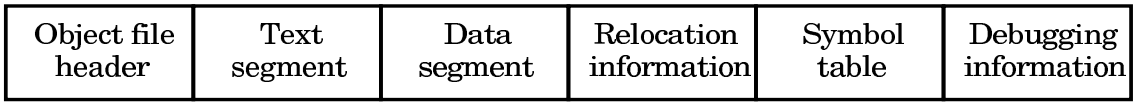
B.1 Assembleur

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 :



- 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 déboggage 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 ou 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 :

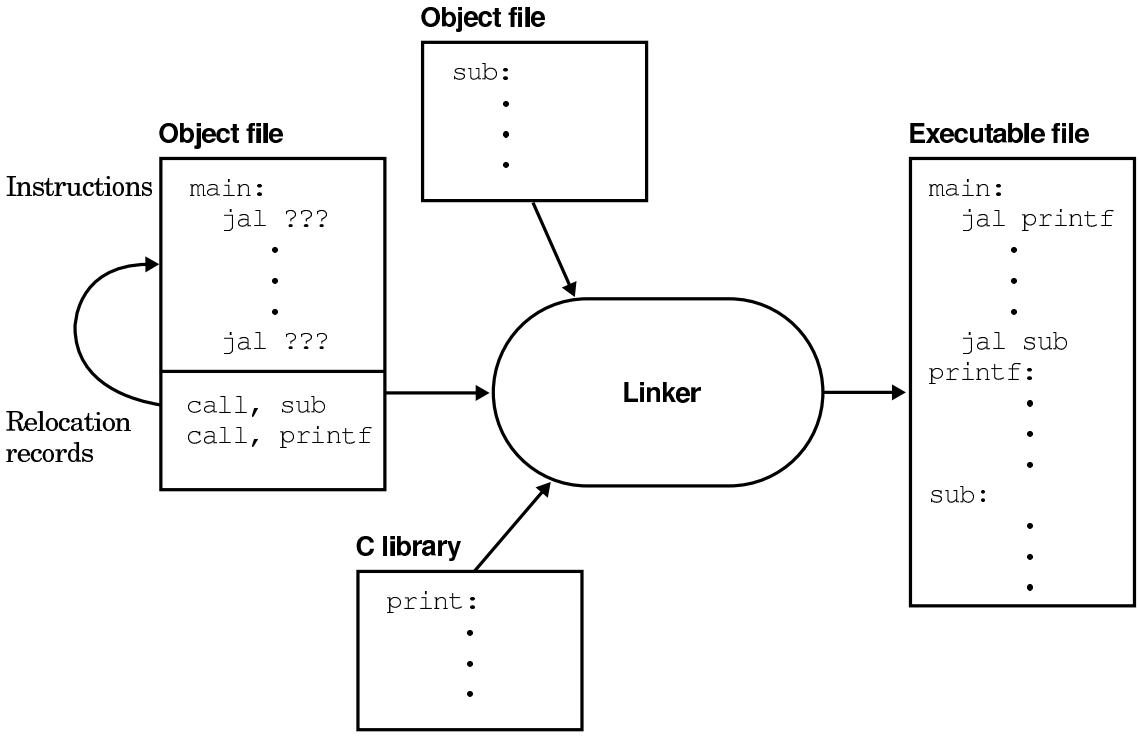
- Les *directives* permettant de décrire des données de manière plus conviviale qu'en binaire, par exemple :
  ```
  .asciiz "la somme est....%n"
  ```
- Les *macros* permettent de réutiliser une séquence d'instructions à plusieurs endroits sans la répéter dans le texte.
- Les *pseudo-instructions* sont 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.

B.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 :



B.3 Chargement

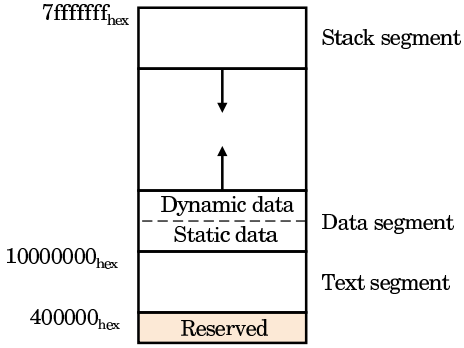
Une fois l'édition de lien réussie, le programme exécutable est stocké sur un système de stockage secondaire, tel qu'un disque dur. Lors de l'exécution sur un système de type Unix, le système d'exploitation charge le programme dans la mémoire vive avant de l'exécuter. Le système effectue donc les opérations 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`.

B.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 connait 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

B.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 designer 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`.

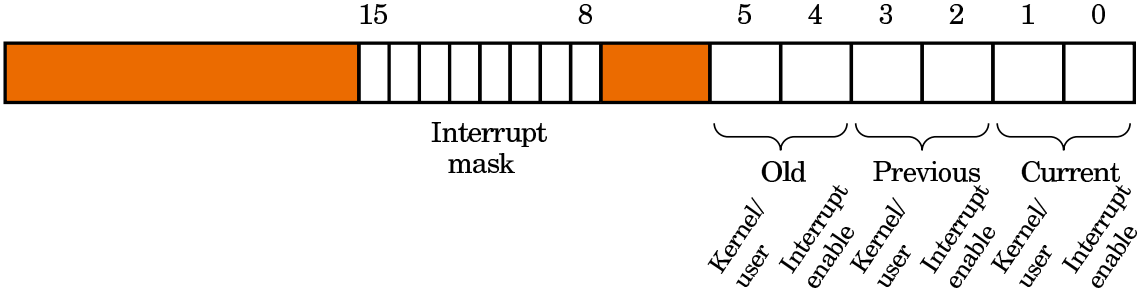
B.6 Exceptions et Interruptions

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

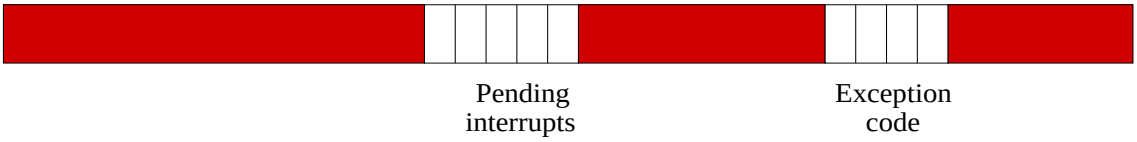
Ces quatre registres sont accédés par des instructions spéciales : `lwc0, mfc0, mtc0`, et `swc0`. Après une exception, le registre `EPC` contient l'adresse de l'instruction qui s'exécutait lorsque l'interruption est intervenue. Si l'instruction effectuait un accès mémoire qui a causé l'interruption, 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 noyeau, 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 regitre `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 reservée
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 80000080_{hex} (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_excp      #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
```

B.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 requètes. 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 $ffff0000_{hex}$. 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.

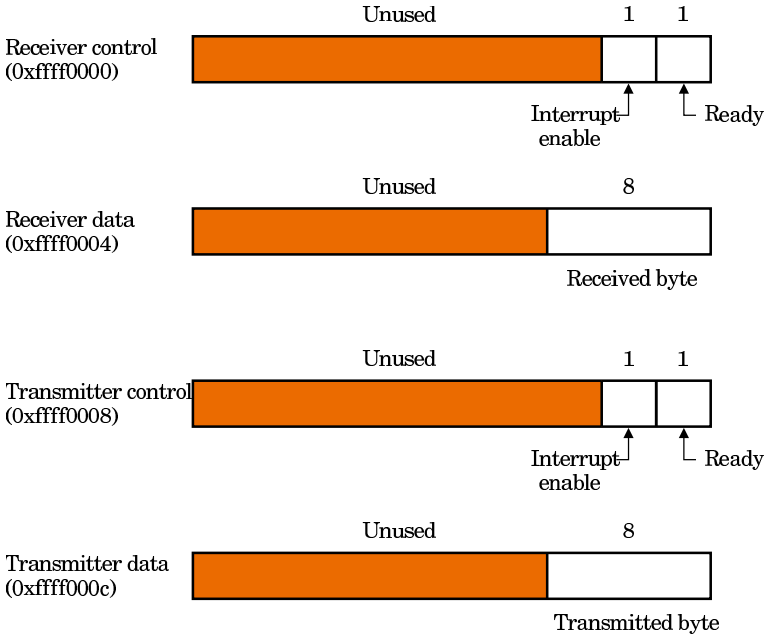
Le deuxième registre du composant est le *registre de donnée du receveur*, il est à l'adresse $ffff0004_{hex}$. Les 8 bits de poids faible de ce registre contiennent le dernier caractère tapé au clavier. Ce registre ne peut qu'être lu par le programme et change dès qu'un nouveau caractère est tapé au 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 précèdent. 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 $fffff000c_{hex}$). 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'execution lors de l'affichage d'un caractère sur la console.



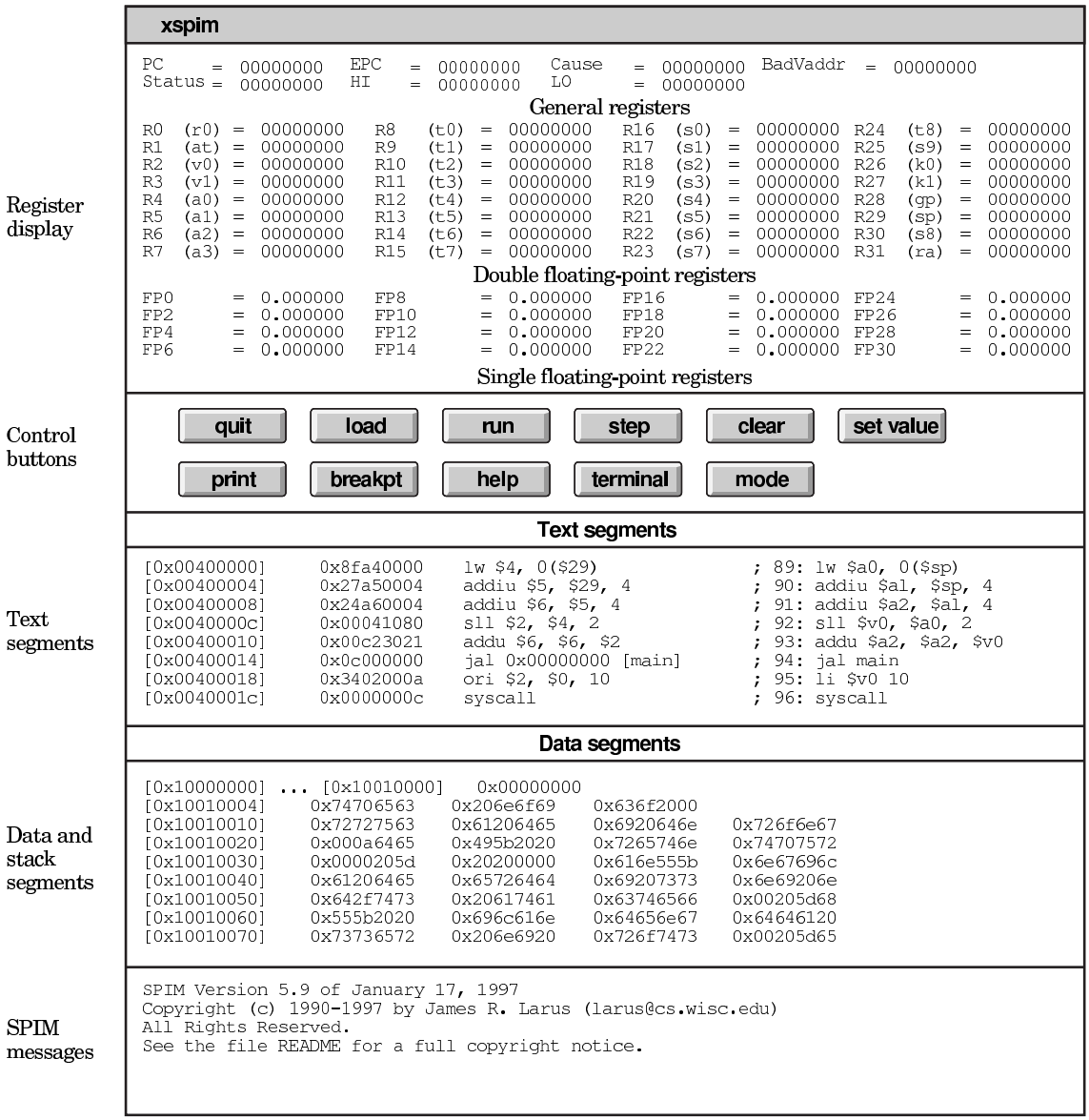


FIG. 4 – Interface X du simulation Spim : simulateur du jeu d'instruction du Mips

B.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.

Debogage `gcc -g ex1.c -o ex1`
Génère un exécutable contenant les informations de debogage 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
 -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 ou elle est définie. Indiquer la section de `man` (section 3) est nécessaire sinon on risque de tomber sur une commande d'un shell.

compilation reciblable Le compilateur `gcc` (ainsi que toutes les commandes ci-dessus) peut être reciblé 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    $0xfffffff0,%esp
  29:   b8 00 00 00 00          mov    $0x0,%eax
  2e:   89 45 f4                mov    %eax,0xfffffff4(%ebp)
  31:   8b 45 f4                mov    0xfffffff4(%ebp),%eax
  34:   e8 00 00 00 00          call   39 <_main+0x19>
  39:   e8 00 00 00 00          call   3e <_main+0x1e>
  3e:   c7 45 f8 00 00 00 00    movl   $0x0,0xfffffff8(%ebp)
  45:   c7 45 fc 00 00 00 00    movl   $0x0,0xfffffffc(%ebp)
  4c:   83 7d fc 64             cmpl   $0x64,0xfffffffc(%ebp)
  50:   7e 02                   jle    54 <_main+0x34>
  52:   eb 15                   jmp    69 <_main+0x49>
  54:   8b 45 fc                mov    0xfffffffc(%ebp),%eax
  57:   89 c2                   mov    %eax,%edx
  59:   0f af 55 fc             imul   0xfffffffc(%ebp),%edx
  5d:   8d 45 f8                lea    0xfffffff8(%ebp),%eax
  60:   01 10                   add    %edx,(%eax)
  62:   8d 45 fc                lea    0xfffffffc(%ebp),%eax
  65:   ff 00                   incl   (%eax)
  67:   eb e3                   jmp    4c <_main+0x2c>
  69:   8b 45 f8                mov    0xfffffff8(%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          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 120). 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 120).