Plan général du cours :

1. Introduction
2. Analyse sémantique
3. Compilation de langages impératifs
4. Production de code intermédiaire
5. Génération de code
6. Optimisation de code
7. Influence des architectures nouvelles sur la production de compilateurs.

1. Introduction

1.1 Les langages de programmation de haut niveau
1.2 Environnement d'un compilateur
1.3 Compilation et interprétation
1.4 Le processus de compilation
 (a) Les buts
 (b) Les sous-tâches
 (c) Les passes
1.5 Problèmes de correction des compilateurs
1.6 Outils pour la production de compilateurs

1.1 Les langages de programmation de haut niveau

Un langage de haut niveau cherche à abstraire la structure et les détails du calculateur sur lequel les programmes sont destinés à être exécutés.

3 grandes classes de langages de haut niveau :

1. Les langages impératifs
 (Algol 60,68, Fortran, Cobol, Pascal,Ada, Modula-2, C).
 Suivent le modèle de calculateur de Von Neumann qui comprend *une unité de calcul (CPU), une mémoire* et *un bus* pour la communication entre la CPU et la mémoire.
2. Les langages fonctionnels
 (Lisp, Hope, Miranda, Haskell, FP).
 - pas de séparation nette entre le monde des instructions et celui des expressions,
 - les noms servent à désigner des expressions et des fonctions, et non les cellules d'une mémoire,
 - les fonctions elles-mêmes peuvent être des arguments ou des résultats d'autres fonctions.

 Un programme fonctionnel est évalué. C'est une *grosse expression* simplifiée jusqu'à obtenir une forme normale.
3. Les langages logiques
 (Prolog et ses différents dialectes).
 Basés sur le mécanisme d'exécution de la logique des Prédicats : la résolution.

Et de nombreux langages pour des applications spécialisées :

- langages de description de matériel : pour spécifier des ordinateurs ou des composants d'ordinateurs,
- langages de commandes de systèmes d'exploitation : pour activer les fonctions du système et les programmes utilisateur, et les faire coopérer,
- langages de spécifications pour traitements de texte et de graphiques : pour décrire des objets de type textuel ou graphique.

1.2 Environnement d'un compilateur

- un éditeur : pour saisir les programmes. De préférence *intelligent*, connaissant la syntaxe du langage de manière à éviter les fautes de frappe.
- un prouveur ou vérificateur de programmes / la spécification d'origine.
- un interpréteur interactif (intégré à l'éditeur) surtout dans le cas où le programme est d'abord écrit dans un langage de spécification.
- un compilateur
  programme source → $\boxed{\text{compilateur}}$ → programme cible
  ↓
  message d'erreurs
- un débogueur interactif, pour permettre la mise au point du programme, (trace, point d'arrêt, interrogation de la valeur des variables).

La création d'un programme cible exécutable peut nécessiter d'autres programmes. En particulier, le programme écrit par l'utilisateur peut être transformé par un préprocesseur ;
Exemple

- les macro définitions
- les include
- les vectorizeurs (Fortran → Fortran vectoriel)
- un paralléliseur

Souvent, le programme cible engendré par le compilateur doit être traduit par un assembleur en code machine.

programme source (squelette)
↓
$\boxed{\text{préprocesseur}}$
↓
programme source
↓
$\boxed{\text{compilateur}}$
↓
programme cible en assembleur
↓
$\boxed{\text{assembleur}}$
↓
code machine translatable
↓
$\boxed{\text{relieur chargeur}}$ ← bibliothèque
↓
code machine

1.3 Compilation et interprétation

Définitions
Compilateur : c'est un traducteur d'un programme P_L (programme source) dans une autre représentation (programme cible).
Les 2 programmes sont sémantiquement équivalents.
Cette traduction permet une exécution efficace avec n'importe quel jeu de données.

Langage cible (ou langage objet) : assembleur ou binaire (ou même langage intermédiaire)

Langage source : celui dans lequel on écrit le programme, en général un langage de haut niveau.

Le coût supplémentaire lié à ce prétraitement du programme sera amorti lors de l'exécution pour un ou plusieurs jeux de données.

Interpréteur : a une signature fonctionnelle

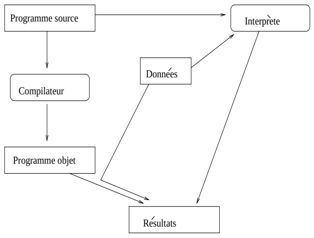
$$I_L \; : \; L \times D^* \rightarrow D^* \cup \{erreur\}$$

L'exécution d'un programme P_L avec les données e et les résultats a est spécifiée par :

$$I_L(P_L, \; e) = a$$

L'interpréteur travaille simultanément sur P_L et e. Il ne se trouve absolument pas préparé à toute construction du programme même en cas d'exécutions répétées.

Comparaison :



Interprète :

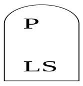
- Avantages : tests immédiats, rapidité de mise au point, convivialité.
- Inconvénients : beaucoup de place mémoire, temps d'exécution important (pas d'optimisation).

Compilateur :

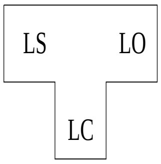
- Avantages : efficacité pour la gestion de la place mémoire, et le temps d'exécution, Meilleure détection des erreurs syntaxiques et sémantiques, optimisation sophistiquée, confidentialité lorsqu'on livre du binaire.
- Inconvénients : plus lourd à mettre en œuvre, moins souple.

1.4 Le processus de compilation

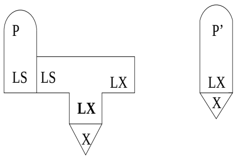
Notations :programme P écrit en LS



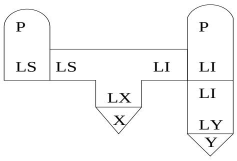
Compilateur LS écrit en LC LS → LO



Compiler un programme écrit en LS sur la machine X puis exécuter sur X

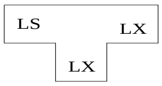


Compilation croisée : compiler sur la machine X et exécuter sur la machine Y.

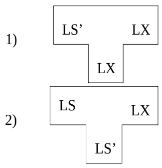


Bootstraping : écrire le compilateur de LS pour la machine X dans un langage moyen qui est un sous-ensemble de LS noté LS'.

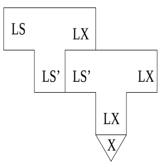
On veut :



On fait :



on compose :



Deux parties :

- *l'analyse* : partitionne le programme source en ses constituants et en crée une représentation intermédiaire. Calcule certaines propriétés sémantiques du programme (sémantique statique).

  Le résultat est soit une représentation appropriée du programme avec ses propriétés statiques, soit des messages d'erreur. Le programme est alors rejeté sauf si les erreurs sont récupérables.

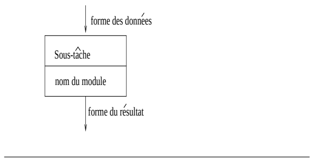
- *la synthèse* : construit le programme cible sémantiquement équivalent à partir de la représentation intermédiaire.

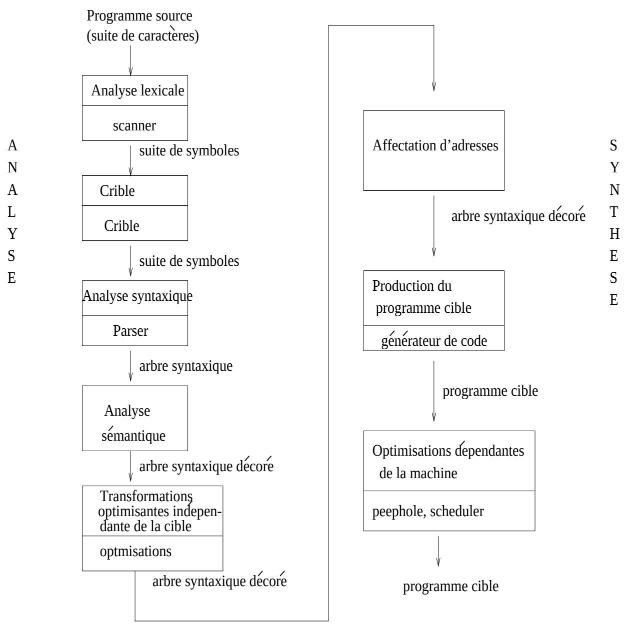
(a) Les buts de la compilation

- Faire toutes les vérifications statiques
  - si le programme est syntaxiquement correct,
  - si toutes les variables sont bien déclarées, faire le contrôle de types des expressions,
  - détecter les instructions, les procédures inutiles,
  - vérifier que les appels de procédures sont cohérents avec les déclarations.
- faire des transformations *source-source* optimisantes en préservant la sémantique,
- traduire : représentation des données et du programme, et génération du code intermédiaire,
- production et optimisation du code machine.

(b) Les sous-tâches

Formalisme :





ANALYSE :

L'analyse lexicale : lecture du programme source sous forme de suite de caractères et structuration en une succession d'unités lexicales du langage appelées symboles (*token*).

Les unités lexicales usuelles :

- nombres entiers,
- nombre flottants,
- booléens,
- caractères,
- chaînes de caractères,
- identificateurs,
- commentaires,
- les signes particuliers, (ex. les signes de ponctuation , ; ...)
- les combinaisons de signes (ex. :=)

ANALYSE :(suite)

Le crible : sa tâche est de reconnaître dans la suite de symboles produite par le scanner,

- ceux qui ont un rôle particulier, (ex. les mots clés du langage)
- les symboles sans importance pour la suite, ex. les séparateurs (les blancs, NL, les commentaires) et qui peuvent être éliminés,
- les parties du texte qui ne font pas partie du programme (ex. les directives de compilation)

De plus, il code certaines classes de symboles de manière non ambigüe, ex. les identificateurs.

ANALYSE :(suite)

L'analyse syntaxique :
Le parser reconnaît la structure syntaxique du programme et en donne une représentation qui permettra aux étapes ultérieures de la traduction de travailler. L'une des représentations possibles est l'arbre syntaxique du programme qui pourra être décoré par des informations sémantiques.

Passage de la suite de mots à la structuration en phrases.

Si on décrit la grammaire de la façon suivante :

PROGRAM	$\rightarrow$	LIST_DECL ptvir LIST_INST pt
LIST_DECL	$\rightarrow$	DECL
	|	LIST_DECL ptvir DECL
DECL	$\rightarrow$	var LIST_ID 2pt TYP
LIST_ID	$\rightarrow$	id
	|	LIST_ID virg id
LIST_INST	$\rightarrow$	INST
	|	LIST_INST ptvir INST
INST	$\rightarrow$	ASSIGN
ASSIGN	$\rightarrow$	id aff EXPR

et si on a la portion de programme :

```
var a,b:int;
a:= 2;
b := a*a+1.
```


ANALYSE :(suite)

L'analyse sémantique :
Vérifier certaines propriétés du langage, ne pouvant être décrites à l'aide d'une grammaire non contextuelle. Ex. la bonne déclaration d'une variable, ou la cohérence de types sont toutes deux des propriétés qui dépendent des règles du langage sur la visibilité et la portée des variables.

Déterminer les propriétés du programme indépendantes du contexte, c'est-à-dire calculables à la compilation.

Ex : b := a*a+1

a est entier, a*a aussi, 1 est un entier,

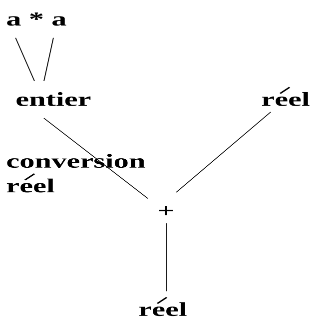
un entier + un entier $\rightarrow$ un entier

b est entier, donc l'instruction est correctement typée.

ANALYSE :(suite)

Si on avait b réel, a entier, et l'instruction

b := a*a+1.2



Il faut décoder + flottant et non entier.

Les opérations arithmétiques sont le plus souvent surchargées.

ANALYSE :(suite)

Optimisations indépendantes de la machine cible :
Des analyses supplémentaires peuvent donner des indications sur les risques d'erreurs à l'exécution ou permettre de découvrir des opportunités d'optimisation. Ex.

- existance possible d'un chemin d'exécution au cours duquel une variable est utilisée sans avoir été initialisée,
- portion de code inutile,
- information sur la valeur des variables

{a = indéf, b = indéf}
a:=2 {a=2, b=indéf}
b:=a*a+1 {a=2, b=5}
peut être remplacé par a:=2, b:=5
Supposons la séquence
a:=2
b:= ((a*a)+1)*c
elle sera remplacée par
a:=2
b:=5*c (c'est le constant folding)

Optimisation locale de code.

Problème NP complet.

S'oppose à la recherche de l'optimalité globale.

SYNTHESE :

Affectation d'adresses :

Interviennent des caractéristiques de la machine cible :

- longueur d'un mot ou d'une adresse ?
- entités directement adressables de la machine ?
- existe-t-il des instructions pour adresser directement et efficacement, ou des parties d'entités directement adressables ?
  Ex. Shift pour transformer une multiplication par une puissance de 2 en décalage.

Ces paramètres déterminent l'affectation des unités aux types élémentaires et conditionnent la possibilité de rassembler plusieurs petits objets (bool ou char) dans des unités plus grandes de la mémoire.

SYNTHESE :(suite)

Attention ! Souvent, restriction d'adressage au niveau des instructions. Ex. contrainte d'alignement : un nombre ne peut être lu ou écrit ou utilisé que s'il est situé sur une frontière de mot.

Exemple : si l'unité d'adressage est l'octet, et que les entiers sont sur 4 octets, pour l'exemple, a sera affecté à l'adresse 0, b à l'adresse 4.

Supposons que l'on ait à affecter successivement a entier, c caractère, b entier,
a sera affecté à l'adresse 0, c à l'adresse 4, et b à l'adresse 8. 3 octets *perdus* si pas d'autre variable caractère.

Il y a souvent en même temps que l'allocation, une production de code intermédiaire.

SYNTHESE :(suite)

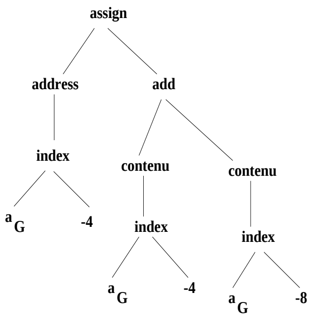
Générateur de code :

Produit les instructions du programme cible.

Pour adresser les variables, utilise les adresses calculées lors de l'étape précédente. Doit aussi pour des raisons d'efficacité du code produit, garder les valeurs des expressions et des variables dans les registres de la machine.

Toutes les machines disposent d'un nombre limité de registres. Le générateur de code doit donc choisir parmi les valeurs pertinentes à stocker dans les registres à un moment donné : allocation de registres.

Doit aussi faire de la sélection de code : meilleur choix d'instructions.
Il y a rarement une seule séquence possible.
Exemple : x := x+y, x(-4), y(-8)



SYNTHESE :(suite)

L'instruction d'addition du MC68000

ADD op1, op2

et a la sémantique suivante op2 $\leftarrow$ op1 + op2 avec comme contrainte sur les modes d'adressage :
un seul des 2 opérandes peut être une référence mémoire.
Il faut donc charger l'un des 2 opérandes dans un registre.

1ère séquence : MOV (A_G, -4), D1
ADD (A_G, -8), D1
MOV D1, (A_G, -4)

2ème séquence : MOV (A_G, -8), D2
ADD D2, (A_G, -4)

Ici, la 2ème séquence est meilleure en temps et en espace.

SYNTHESE :(suite)

Supposons une machine cible avec les instructions suivantes :

	source, destination	
LOAD	adr, reg	
STORE	reg, adr	
LOADI	int, reg	met un entier dans un registre
ADDI	int, reg	
ADD	adr, reg	
MUL	adr, reg	adr * reg $\leftarrow$ reg

Soient $R_1, \ldots,\ R_N$ les registres de la machine,
et soit à traduire a := 2; b := a*a+1

LOADI	2, R_1
STORE	R_1, 0
LOAD	0, R_1
MUL	0, R_1
ADDI	1, R_1
STORE	R_1, 4

SYNTHESE :(suite)

Optimisations dépendantes de la machine cible :

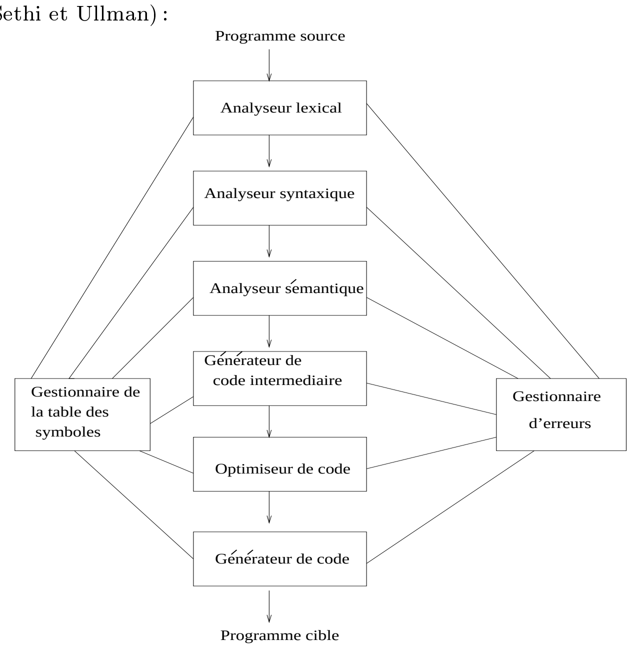
optimisations locales (*à lucarne ou peephole*)
- éliminer les instructions inutiles,
- remplacer les instructions générales par des instructions plus spécifiques adaptées à ce cas particulier. Exemple :
  MOV 0, R_1 remplacée par CLR R_1

Dans l'exemple précédent, la 3ème instruction est inutile.
Allocation *au hasard* $\longleftrightarrow$ allocation intelligente.

Souvent, il existe une instruction incrément qui ajoute la valeur 1 à un registre rapidement.
On remplacerait ADDI 1, R_1 par INC R_1 (plus rapide).

SYNTHESE :(suite)

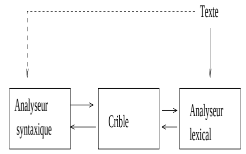
On peut aussi voir cette structuration en tâches comme dans le *Dragon* (Compilateurs, principes, techniques et outils, Aho, Sethi et Ullman) :



(c) Les passes d'un compilateur

Une passe = un parcours du texte source.

Il est courant que plusieurs phases soient regroupées en une passe et que leurs activités soient entrelacées durant l'exécution de cette passe.
Ex. on peut regrouper en une passe l'analyse lexicale, le crible et l'analyse syntaxique. Dans ce cas, l'analyseur syntaxique appelle le crible comme un sous-programme. Un seul module réalise ainsi les 3 premières tâches.



Si

- l'analyse sémantique n'est pas trop compliquée,
- pas d'optimisation de code, alors

on peut visiter **une seule fois** l'arbre pour produire le code.
compilateur mono-passe.
Dans ce cas, le flot d'unités lexicales produit par l'analyse lexicale est traduit directement en code intermédiaire. L'analyseur syntaxique est **maître**. Il tente de découvrir la structure grammaticale du texte (en appelant au besoin le scanner) et au fur et à mesure, il fait appel au générateur de code intermédiaire pour qu'il effectue l'analyse sémantique et produise une portion de code.

1.5 Problèmes de correction des compilateurs

Un compilateur C est une fonction LS $\rightarrow$ LO. Sa correction est définie relativement à un domaine sémantique (DS pour domaine source et DO pour domaine objet).

La correction consiste à prouver la commutativité du diagramme :

$$\begin{tikzcd} LS \arrow[r, "C"] \arrow[d, "S.Sem"'] & LO \arrow[d, "O.Sem"] \\ DS \arrow[r, "Rep"] & DO \end{tikzcd}$$

Prouver un compilateur est une tâche ardue. La seule possibilité est de le rendre modulaire et d'introduire des étapes dans le diagramme.

Cette complexité justifie le recours à des méthodes de spécification de compilateurs liées aux méthodes de description de la sémantique du langage source.

1.6 Outils pour la production de compilateurs

Sous-tâche du compilateur	mécanisme de spécification	type d'automate
Analyse lexicale	expression régulière	automate d'états finis déterministe
Analyse syntaxique	grammaire non contextuelle	automate à pile déterministe
Analyse sémantique	grammaire attribuée	
Transformation optimisante	transformation d'arbre	traducteur d'arbre
Sélection de code	grammaire d'arbre régulière	automate fini d'arbre

D'autres tâches utilisent d'autres outils :

- moteur d'analyse de flot de données (résolution d'équations)
- allocation de registres (coloriage de graphes)

Générateur de compilateur : pour chaque sous-tâche, spécifier formellement et utiliser une technique de génération à partir de cette spécification.

2. Analyse sémantique

2.1 Introduction

Construction d'analyseurs syntaxiques avec Lex et Yacc :

- Fabrication du scanner : écrire la spécification du scanner dans scan.l
  lex scan.l : Ce programme fabrique un fichier lex.yy.c qui est le programme de l'analyseur lexical. Plus précisément, la fonction d'appel de l'analyseur lexical yylex() retourne le code de l'unité lexicale suivante dans la chaîne d'entrée.
- Fabrication du parseur : écrire la spécification du parser dans gram.y (en indiquant le lien avec le scanner).
  yacc gram.y : ceci produit un fichier y.tab.c qui contient le parseur. C'est la fonction yyparse() qui retourne un entier
  - 0 si aucune erreur détectée
  - 1 sinon
- Appeler le parseur dans l'environnement d'un programme C

Le programme principal minimum :

```
#include <stdio.h>
main()
{int i;
printf("debut\n");
i= yyparse();
printf("fini i=%d\n",i);
}

yyerror(s)
char *s;
{ fprintf(stderr,"%s\n",s);        appele par yyparse s'il y a
                                   des erreurs detectees
}
yywrap()
{
return(1);}                        appele par le scanner pour savoi
```

si la fin de fichier est une fin de fichier ou s'il faut passer à un autre fichier. Par défaut, 1 signifie fin de données.

yyparse appelle itérativement yylex à chaque fois qu'il a besoin de décoder un token.

list $\rightarrow$ expr

expr $\rightarrow$ NUMBER | expr '+' expr | expr '*' expr

Tous les terminaux sont décrits dans lexgram.l

```
NUMBER [0-9]
%%
{NUMBER}+ {sscanf(yytext,"%d",&yylval);return NUMBER;}
[+] {return PLUS;}
[*] {return MULT;}
[.] {printf("stop");return(POINT);}
%%
```

On décrit la grammaire :

```
%token NUMBER
%token PLUS
%token MULT
%token POINT

%left PLUS
%left MULT
%start list
%%
list: expr POINT { printf("d\n",$1);};

expr : NUMBER {$$=$1;printf("%d\n",$1); }
     | expr PLUS expr {$$=$1 + $3;}
     | expr MULT expr {$$=$1 * $3;};
%%
#include "lex.yy.c"
```

Construction de l'analyseur complet :
cc env.c y.tab.c -o analyse

Soit p le programme :
35 + 48 .

analyse < p fera l'analyse de p et l'exécution donnera
debut
stop fini i=0

Pour retourner la valeur de l'expression, on associe une action à chaque règle. A chaque réduction, l'action sera exécutée.

Pour chaque nombre lu, on lui associe sa valeur numérique et on la rend visible du parseur. Yacc et Lex partagent la même variable globale **yylval** Dans lexgram.l, entre les accolades, on écrit le morceau de programme C qui charge la valeur dans yylval
sscanf(yytext,"%d",&yylval); return(NUMBER)
Dans gram.y, on donne une valeur à $$ à partir des valeurs $$x$ des éléments en partie droite des règles.

2.2 Le rôle de l'analyse sémantique :

Certaines propriétés d'un langage de programmation ne peuvent être décrites à l'aide d'une grammaire non contextuelle, comme vérifier qu'une variable est bien déclarée, ou vérifier la cohérence de types.

Ces propriétés dépendent du langage sur la visibilité et la portée des variables.

Règles de portée : dans quelle partie du programme les déclarations d'identificateurs sont valides ?

Règles de visibilité : dans quelle partie du programme, dans la portée d'un id, cet id est-il visible ?

Terminologie : identificateur = symbole (token) utilisé dans un programme pour désigner un objet.

Un objet peut être
- une variable,
- une constante,
- un type
- une procédure (une fonction)
- un prédicat
- un foncteur

Propriétés sémantiques statiques : communes à toutes les exécutions, et calculables dès la compilation.

Une propriété (non contextuelle) relative à une construction d'un langage de programmation est qualifiée de **sémantique statique** si :
1. pour chaque occurrence de cette construction, la valeur de la propriété sera la même dans toute exécution de la construction. Ex. le type d'une expression est le même pour toutes les valeurs que l'on pourra obtenir dynamiquement pour cette expression. (c'est *plus exactement*, le type le plus général).
2. cette propriété peut être calculée pour chaque occurrence de la construction dans un programme correct. Ex. dans l'exemple du typage, le type peut être calculé pour chaque expression d'un programme correct, aussi bien pour un langage fortement typé que pour un langage polymorphe.

Propriétés sémantiques dynamiques : ne sont en général connues qu'à l'exécution. Ex. valeur d'une variable.

La **déclaration** d'un id l'introduit comme désignation d'un objet (souvent contenu dans la déclaration elle-même).

Exemple : déclaration de type, le type est décrit dans la déclaration.

L'occurrence d'un id dans une déclaration constitue une **définition**.

Les autres sortes d'occurrences sont des **utilisations**.

Dans tout langage de programmation, il existe des constructions qui réduisent la portée des id : **constructions de portée (scope)**.

Exemple : déclaration de procédure, les blocs, les paquets, les modules.

On appelle **blocs** les occurrences de construction qui restreignent la portée dans les programmes.

Le **type** d'un objet indique quelles sont les opérations valides pour cet objet.

Quand les langages de programmation autorisent plusieurs déclarations d'un même id, pour désigner des objets différents, il faut une règle pour préciser quelle définition s'applique à une utilisation donnée : **Règles de validité et de visibilité**.

Domaine de validité (ou portée) : d'une définition d'un id x est la partie du programme dans laquelle une utilisation de x se rapporte à cette définition.

Il faut associer à toute utilisation d'un id la définition à laquelle il correspond : c'est **la tâche de reconnaissance des identificateurs**.

Pb particulier : langages qui autorisent la surcharge d'identificateurs.

Validité :
Cobol ne permet aucune imbrication de constructions de portée : tous les id sont partout visibles et partout valides.

Fortran et C n'autorisent qu'une profondeur d'imbrication. Les id définis dans un bloc ne sont visibles que dans ce bloc.
Les id du programme principal sont visibles partout sauf dans les procédures qui redéfinissent cet id.

Pascal et Ada, les langages fonctionnels en général permettent l'imbrication récursive des blocs sur une profondeur arbitraire.

Visibilité : si la définition d'un id x est globale pour un bloc courant, elle peut être recouverte par une autre déclaration de x. Elle cesse alors d'être directement visible.

En général, il existe des instructions qui permettent de rendre visibles ces id en étendant leur nom avec le nom du bloc où ils ont été définis.

Exemple : with de Pascal, directive use de Ada.

Vérification des conditions de contexte :

- vérifier que la déclaration est correcte,
- vérifier la consistance de types.

Comment fait un compilateur ?

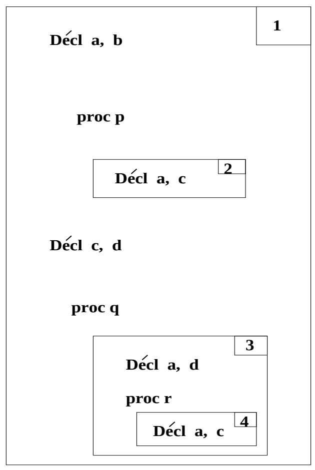
On suppose : langage avec constructions de portée emboîtées mais sans module ni surcharge (au sens Ada avec partie privée-partie publique)

Dans ce cas simplifié, à toute utilisation correspond une définition unique. Les seul pb : trouver la relation entre utilisations et définitions et vérifier que ce lien peut être déterminé sans ambiguïté.

Pour représenter cette correspondance, plusieurs possibilités.

Traditionnellement, le compilateur crée une table des symboles, où pour chaque id, on range ses définitions avec toutes les infos correspondantes.

La TDS est généralement structurée en blocs parallèlement à la structure des blocs du programme.



La TDS n'est pas le résultat de la reconnaissance des identificateurs, elle sert seulement de support à celle-ci ; en effet, elle a pour but de ranger en mémoire, pour une **utilisation** d'un id l'information sur la définition correspondante, c'est-à-dire, soit

1. un pointeur vers le nœud de la déclaration ou
2. l'adresse de l'entrée de la TDS correspondant à la déclaration
3. ou l'information déclarative elle-même.

Avec (1) et (3), on peut faire disparaître la TDS à la fin de cette analyse. C'est l'arbre syntaxique enrichi des informations contextuelles qui reste l'unique structure de données.

Avec (1) en plus, toutes les utilisations d'une même définition réutiliseront la même information.

On choisit l'alternative (1).

Quelles sont les opérations que doit offrir la TDS ? Lorsqu'on rencontre une déclaration :

- inscrire le symbole déclaré dans la table
- mémoriser un pointeur vers le nœud de l'arbre syntaxique concerné

Cette déclaration se trouve dans un bloc, il faudra donc des opérations pour noter les ouvertures et les fermetures de bloc.

La fermeture doit supprimer de la table les entrées correspondant aux décalarations du bloc que l'on ferme.

La table ne contient à chaque instant que les entrées pour les déclarations des blocs encore ouverts.

Lorsque le module de reconnaissance trouve une utilisation, il cherche dans la table conformément aux règles de validité et de visibilité.
S'il trouve, il recopie le pointeur qu'elle contient sur la *bonne* déclaration dans le nœud de l'arbre correspondant à l'utilisation.

On doit avoir la liste des opérations suivantes :

- *create_symb_table* : crée une table vide
- *enter_block* : enregistre l'ouverture d'un nouveau bloc
- *exit_block* : restaure la table dans l'état où elle était avant le dernier enter_block
- *enter_id(id, decl_ptr)* : crée une entrée dans la table pour *id*, cette entrée contient un pointeur vers la déclaration.
- *search_id(id)* : recherche la définition de *id* et renvoie le pointeur vers la déclaration si elle existe.

Les deux dernières opérations travaillent relativement au dernier bloc ouvert : le bloc courant.

Syntaxe concrète, syntaxe abstraite

Syntaxe concrète : représentée par l'arbre syntaxique conformément à la grammaire non contextuelle qui définit le langage.
Cependant, cet arbre contient des informations qui n'ont plus d'importance pour la suite (ex. les symboles terminaux significatifs pour l'analyse syntaxique mais sans importance pour la suite, comme les mos clé).

Pour exprimer les précédences d'opérateurs, on peut être amené à introduire des non terminaux emboîtés, cela n'a plus lieu d'être quand la structure est reconnue.

Syntaxe abstraite : en général, celle sur laquelle le compilateur travaillera. Elle ne contient que l'essentiel de la structure syntaxique et elle repère simplement les constructions intervenant dans le programme avec
leur profondeur d'emboîtement.

Règle de validité : comme en Ada : une définition d'identificateur n'est valide qu'à partir de la fin de sa déclaration. On peut introduire une déclaration de procédure ne contenant que sa spécification. (La procédure est valide même si elle n'est pas encore définie.)

Procédure analyse_decl(k:nœud)
  proc analyse_sousarbres(racine : nœud); ;
  début
   pour i:=1 **à** #descs(racine) **faire**
    analyse_décl(racine.i)
   fait   **fin ;**
début
dans le cas où k désigne :

- un bloc : enter_block
  analyse_sous_arbres(k)
  exit_block
- une déclaration : analyse_sous_arbres(k)
  pour chaque id déclaré ici **faire**
   enter_id(id, ↑k)
- une utilisation de id : mémoriser (search_id(id), k)
   on mémorise le pointeur vers la définition de id
- autre : **si** k n'est pas une feuille **alors**
  analyser_sous_arbre(k)

fin

Ada : pour décl, on traite récursivement toutes les déclarations avant d'écrire les entrées des décl locales.

Pascal : c'est différent. Dès qu'un id apparaît, il est utilisable (*récursivement*). Il faut donc réécrire le cas décl

2.3 Implantation de la table des symboles :

Une bonne implantation doit garantir qu'à tout instant, la fonction search_id renvoie l'entrée qui correspond bien aux règles de visibilité.

1ère solution : une liste linéaire.
enter_block decl1 ... decln enter_block decl1 ...
search_id parcourt la liste en partant de la fin. exit_block efface toutes les déclarations jusqu'au enter_block le plus proche.

Ceci a un comportement de pile. On pourrait bien sûr organiser la liste en pile.

Mais, search_id sera très couteûx s'il y a beaucoup d'identificateurs déclarés (complexité linéaire en nb de décl).

Mieux : complexité logarithmique.

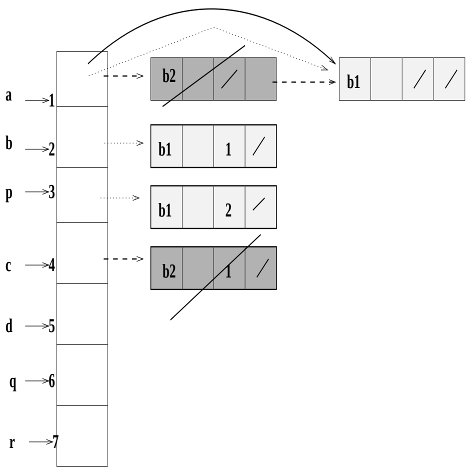
Hyp : chaque id sera sûrement analysé plusieurs fois. On dispose les entrées pour un bloc dans un arbre binaire de recherche : la recherche commence par le bloc courant, puis le bloc englobant etc...

On chaîne ensemble dans une même liste linéaire, toutes les entrées correspondant au même identificateur. L'identificateur sert à indexer un tableau où on aura un pointeur vers la dernière entrée introduite pour cet id.

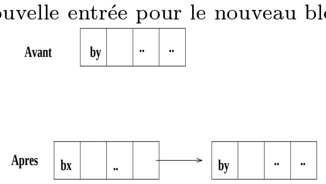
D'autre part, on chaîne toutes les entrées appartenant au même bloc de façon à faciliter le exit_block.

Un pointeur associé au bloc pointe vers la tête de chaîne (des entrées) et ces têtes de chaîne peuvent être organisées en pile.

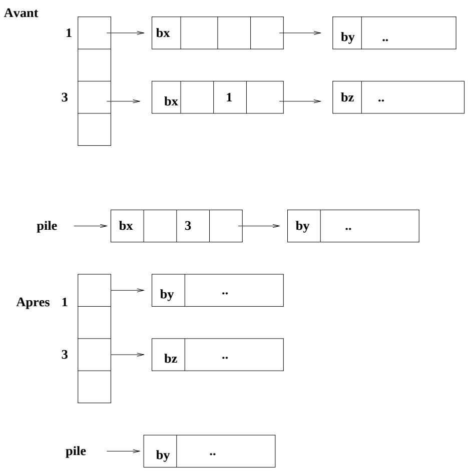
Exemple :



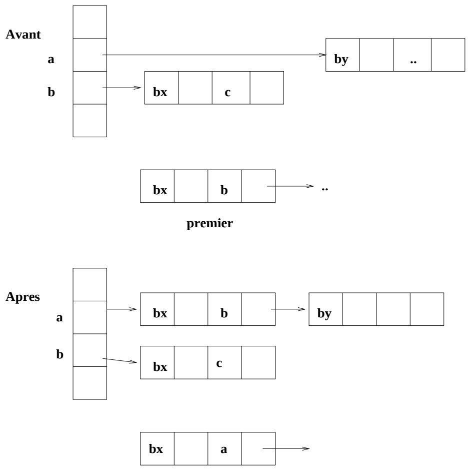
procédure enter_block :
 empile une nouvelle entrée pour le nouveau bloc x



procédure exit_block :
 pour chaque entrée de déclaration du bloc courant
  faire supprimer l'entrée
 dépiler l'entrée du bloc



enter_id(id)
si il existe déjà une entrée pour id dans ce bloc
alors erreur
sinon créer une nouvelle entrée
mettre prec = valeur de premier dans le sommet de pile
mettre suiv = le premier de la chaîne pour cet id
changer la valeur de premier au sommet de pile par l'adresse de cette nouvelle entrée



Une petite conclusion : Pour illustrer le rôle de l'analyse sémantique, on a choisi la tâche de reconnaissance des identificateurs et on a décrit une technique pour la résoudre. D'autres tâches peuvent aussi se faire comme le test de types en une passe ascendante sur l'arbre du programme (ici, plus particulièrement les expressions).

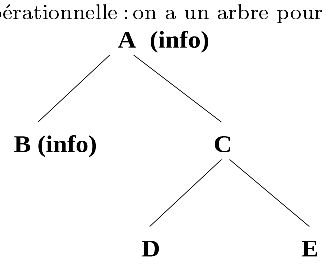
2 aspects pour le rôle de l'analyse sémantique :

- le calcul, la mémorisation d'informations,
- la transformation de l'arbre, (et pour sous-rôle, la traduction en code intermédiaire).

Une méthode de spécification parfaitement adaptée dans ces cas de *traduction dirigée par la syntaxe* est celle des grammaire attribuées.

2.4 Les grammaires attribuées :

Idée intuitive opérationnelle : on a un arbre pour un programme correct.



et on veut accrocher de l'information aux nœuds de l'arbre (décorer l'arbre). Cette info dépend des fils ou du père (et frères gauches) du nœud.

On spécifie pour chaque règle de production de la grammaire le *décor* à accrocher à chaque symbole de la règle ainsi que la façon de le calculer.

Ce décor est appelé *attribut*, considéré comme un support d'information sémantique.

Une grammaire attribuée permet aussi d'indiquer les dépendances fonctionnelles entre les valeurs des occurrences d'attributs dans les productions de la grammaire.
Cette dépendance est une notation de calcul qui détermine comment calculer la valeur d'une occurrence d'attribut en fonction des valeurs d'autres occurrences d'attributs dans la même production.

Il faut des conditions pour garantir qu'il est possible d'évaluer tous les exemplaires d'attributs dans l'arbre syntaxique d'un programme correct.

Définition : Grammaire attribuée
Soit $G = (N,\ T,\ P,\ S)$ une grammaire non contextuelle.
Soit $p \in P,\ p\ :\ X_0 \to X_1 \dots X_{n_p},\ X_i \in N \cup T,\ 0 \le i \le n_p$

Une grammaire attribuée GA sur G comprend :

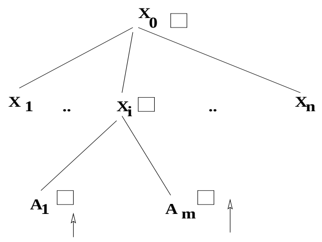
- pour tout symbole $X \in N \cup T$, deux ensembles $Her(X)$ l'ensemble des attributs hérités et $Syn(X)$ l'ensemble des attributs synthétisés. tels que $Her(X) \cap Syn(X) = \emptyset$.
  On note $Attr(X) = Her(X) \cup Syn(X)$.

  Si $a \in Attr(X_i)$ alors a présente une occurrence dans la production p pour l'occurrence de X_i notée a_i

$$Her = \bigcup_{X \in N} Her(X)\,;\ Syn = \bigcup_{X \in N \cup T} Syn(X)\,;\ Attr = Her \cup Syn$$

- un domaine de valeurs D_a est fixé pour chaque attribut $a \in Attr$

Attributs synthétisés :

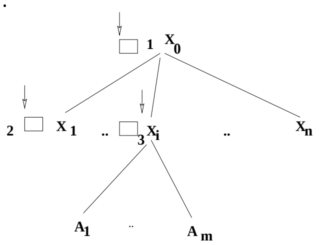


Les attributs synthétisés sont évalués en faisant un parcours **bas-haut, gauche-droite** de l'arbre syntaxique.

Exemple : parfaitement adaptés pour calculer le type d'une expression.

Donne des systèmes qui fonctionnent efficacement (ex. Yacc + gestion des variables globales).

Attributs hérités :



3 est hérité de $X_0,\ X_1, \dots X_{i-1}$

Dans une production $X_0 \to X_1 \dots X_i \dots X_n$, toutes les occurrences d'attributs $a \in Her(X_i),\ i \in 1 \dots n$ ainsi que $b \in Syn(X_0)$ sont des occurrences de définitions. Les autres, c'est-à-dire $a' \in Syn(X_i),\ i \in 1 \dots n$ et $b' \in Her(X_0)$ sont des occurrences d'utilisation.

Une grammaire attribuée est sous *forme normale* lorsque tous les arguments des règles sémantiques sont des occurrences d'utilisation.

Exemple1 :

On veut spécifier le calcul de type pour des expressions où les opérateurs sont $+,\ -,\ *,\ /$ et $\div$ et les variables ou constantes de type integer ou real.

On suppose que le type est déjà calculé pour les terminaux (se fait en collaboration avec le scanner)

On définit une fonction f à 3 arguments

$$f\ :\ \text{op}\ \times\ \text{type}\ \times\ \text{type}\ \to \text{type} \cup \{\text{erreur}\}$$

	int × int	real × int	int × real	real × real
+	int	real	real	real
-	int	real	real	real
*	int	real	real	real
/	real	real	real	real
÷	int	erreur	erreur	erreur

Exemple 1 : GA1

$E \rightarrow E_1$ Aop T
E_0.type = f(Aop.op, E_1.type, T.type)

$E \rightarrow T$
E.type = T.type ← règle de recopie

$T \rightarrow T_1$ Mop F
T_0.type = f(Mop.op, T_1.type, F.type)

$T \rightarrow F$
$F \rightarrow P$
$P \rightarrow \%const$
P.type = $\%const$.type

$P \rightarrow \%var$
P.type = $\%var$.type

$P \rightarrow (E)$
P.type = E.type

Aop → '+'
Aop.op = '+'

Aop → '-'
Aop.op = '-'

Aop → '*'
Aop.op = '*'

Aop → '/'
Aop.op = '/'

Aop → '÷'
Aop.op = '÷'

Non-terminaux :
E, T, F, P, Aop, Mop

Attributs synthétisés :
type : défini sur E, T, F, P
domaine de valeurs = {int, real}

op : défini sur Mop, Aop
domaine de valeurs = {'+', '-', '*', '/', '÷'}

Hypothèse : $\%const$ et $\%var$ ont aussi l'attribut type dont la valeur est fournie par le scanner.

Remarque : la majeure partie des règles sémantiques sont de simples recopies. On calcule en un point, puis on *véhicule* l'information jusqu'à un autre point de calcul.

Pour cela, on modifie les règles de GA1 de façon à intégrer la notion de type attendu d'une expression.
On introduit : typeoblig, hérité pour E, T, F, P
domaine de valeurs = {int, nonspec}

f2	typeoblig-op1	typeoblig-op2
'÷'	int	int
'*'	nonspec	nonspec
'/'	nonspec	nonspec

lire : f2('÷', 1) = int   f2('÷', 2) = int
f2('*', 1) = nonspec ...

{E_1.typeoblig = E_0.typeoblig ; T.typeoblig = E_0.typeoblig}
$E \rightarrow E_1$ Aop T
E_0.type = f(Aop.op, E_1.type, T.type)

$E \rightarrow T$
{T_1.typeoblig = f2(Mop.op,1) ; F.typeoblig = f2(Mop.op,2)}
$T \rightarrow T_1$ Mop F
T_0.type = f(Mop.op, T_1.type, F.type)

$T \rightarrow F$
$F \rightarrow P$
$P \rightarrow \%const$
{si $\%const$.type = P.typeoblig ou si P.typeoblig = nonspec
alors P.type = $\%const$.type
sinon écrire erreur
la const doit être du type P.typeoblig }

$P \rightarrow \%var$
{si $\%var$.type = P.typeoblig ou si P.typeoblig = nonspec
alors P.type = $\%var$.type
sinon écrire erreur
la var doit être du type P.typeoblig }

...

Remarque : les règles sémantiques ne doivent pas faire apparaître de dépendances mutuelles entre les attributs : pas de cycles tels que de façon infinie, le calcul de A attende le calcul de B qui attend celui de A.

A partir de cette spécification de la grammaire attribuée, on veut générer automatiquement un évaluateur d'attributs, i.e. qui détermine l'ordre d'évaluation des attributs dans l'arbre.

Ce qui est difficile : faire le bon choix d'attributs pour effectuer la tâche attendue, en particulier lorsqu'il y a beaucoup d'attributs à gérer.

Un deuxième exemple :
La grammaire suivante décrit la représentation binaire de nombres décimaux. On l'attribue de telle façon qu'on obtienne la conversion en nombre décimal du nombre binaire reconnu.

Grammaire attribuée Bin_to_Dec :
non terminaux : {N, BIN, BIT}
attributs :

1. lg synthétisé associé à BIN pour domaine int
2. val synthétisé associé à N, BIN, BIT sur real
3. rg hérité associé à BIN, BIT sur int

$N \rightarrow \{BIN_1.rg = 0\,;BIN_2.rg = \text{-}(BIN_2.lg{+}1)\}$ BIN . BIN
$\{N.val = BIN_1.val + BIN_2.val\}$

$N \rightarrow \{BIN_1.rg = BIN_0.rg + 1\,;BIT.rg = BIN_0.rg\}$ BIN BIT
$\{BIN_0.lg = BIN_1.lg + 1\}$
$\{BIN_0.val = BIN_1.val + BIT.val\}$

$BIN \rightarrow \{BIT.rg = BIN.rg\}$ BIT
$\{BIN.lg = 0\}$
$\{BIN.val = BIT.val\}$

$BIT \rightarrow 1$
$\{BIT.val = 2^{BIT.rg}\}$

$BIT \rightarrow 0$
$\{BIT.val = 0\}$

2.5 Génération d'évaluateurs d'attributs :

1. Ce qu'on veut évaluer, ce sont les instances d'attributs dans les arbres syntaxiques.

2. étant donnée une grammaire attribuée, produire un évaluateur qui puisse effectuer (1).

La grammaire attribuée définit un système d'équations pour chaque arbre syntaxique de la grammaire sous-jacente. Les inconnues de ce système sont les instances d'attributs des nœuds de l'arbre.

2.5 Génération d'évaluateurs d'attributs :

1ère idée : pour un arbre syntaxique,

- c'est un très gros système d'équations
- s'il n'est pas récursif, on peut le résoudre par un processus d'élimination. A chaque étape de l'élimination, il faut rechercher la prochaine instance d'attribut ne dépendant que d'instances déjà calculées et en calculer la valeur.
- processus dynamique. Pas très efficace.

2ème idée : les règles sémantiques spécifient les dépendances entre les occurrences d'attributs des productions. (une occurrence d'attribut a_i dépend d'une occurrence b_j si b_j est un argument de la règle sémantique pour a_i)

Le générateur associé analyse les dépendances dans la grammaire attribuée pour déterminer **statiquement** l'ordre d'évaluation.
Il produira un évaluateur d'attributs. (il ne cherchera pas forcément la prochaine instance d'attribut à évaluer, mais déterminera l'ordre d'évaluation selon **une règle de visite**.

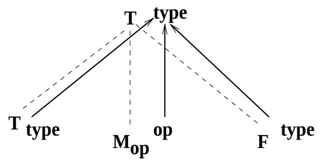
Dépendances d'attributs : Soit p une règle de production ;
$Oc(p)$: ensemble des occurrences d'attributs de p.

Dépendance locale dans la production :
La relation de dépendance locale $D_p(p) \subset Oc(p) \times Oc(p)$ dans p est définie comme suit :

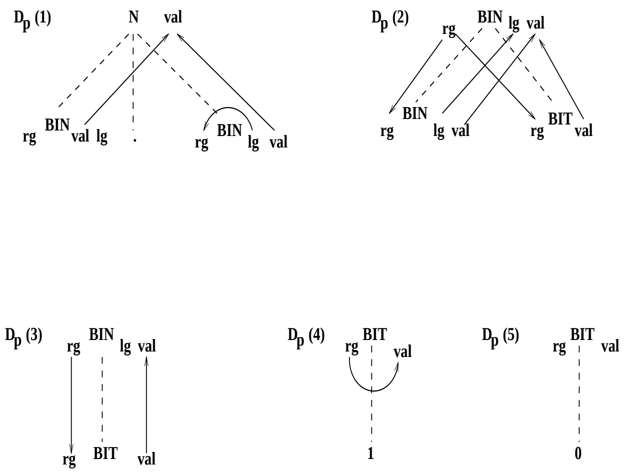
$$b_j \ D_p(p) \ a_i \Longleftrightarrow a_i = f_{p,a,i}(\ldots, b_j, \ldots)$$

pour une règle sémantique de p (a_i dépend de b_j). On représente cette relation par un graphe.

Ex. 3 : $T_0 \to T_1 \ M_{op} \ F \ \{T_0.type := f(M_{op}.op, T_1.type, F.type)\}$
$D_p(3)$



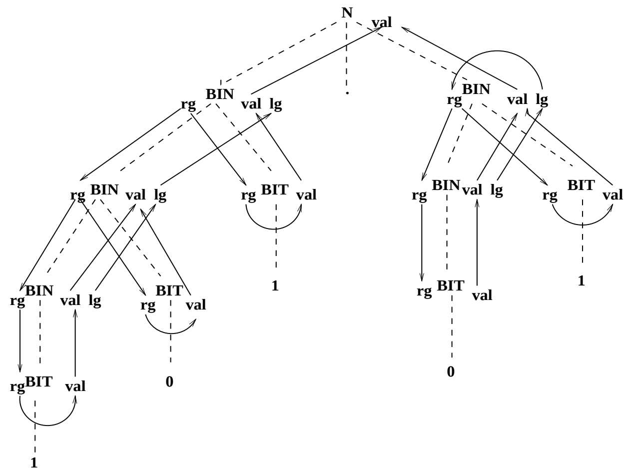
Pour la grammaire Bin_to_Dec, l'ensemble des graphes de dépendance locale :



Les dépendances locales aux productions induisent des dépendances entre les instances d'arguments dans les arbres de la grammaire.

Graphe de dépendances d'un arbre :
Soit t un arbre de la grammaire non contextuelle. On obtient $D(t)$ sur les instances d'attributs de t en assemblant les graphes de dépendances locaux pour les productions appliquées à t.

Exemple : arbre syntaxique avec graphe de dépendances pour le mot 101.01



Intuitivement, s'il y a des cycles dans ce graphe, on ne pourra pas calculer les valeurs des attributs.

Définition : une grammaire attribuée est non circulaire lorsque
$\forall\ t \in$ grammaire, $D(t)$ n'est pas cyclique.
Une GA est bien formée ssi elle est non circulaire.

Définition : Un tri topologique d'un graphe orienté acyclique est un ordonnancement quelconque $m_1,\ m_2,\ \ldots,\ m_k$ des sommets du graphe tel que, pour tous les sommets du graphe, pour tout arc, le sommet de départ apparaisse avant le sommet d'arrivée dans l'ordonnancement.

Exemple : dans l'arbre p
19 - 20 - 21 - 22 - 23 - 6 - 5 - 4 - 3 - 7 -
24 - 25 - 26 - 27 - 28 - 29 - 30 - 15 - 14 -
13 - 12 - 11 - 33 - 32 - 31 - 19 - 18 - 17 - 16 -
10 - 9 - 8 - 2 - 1 convient

mais aussi 19 - 20 - 24 - 25 - 26 - 27 - ...

Tout tri topologique d'un graphe de dépendances donne un ordre valide pour l'évaluation des règles sémantiques associées aux nœuds de l'arbre syntaxique.

Plusieurs méthodes pour évaluer les règles sémantiques :

1. **Méthodes fondées sur l'arbre syntaxique**
   A la compilation, ces méthodes déterminent un ordre d'évaluation à partir du graphe de dépendances construit sur l'arbre syntaxique pour **un** texte d'entrée.
   Echec s'il y a un cycle.
2. **Méthodes fondées sur les règles sémantiques**
   A la construction du compilateur, on analyse les règles sémantiques.
   Pour chaque règle de production, l'ordre d'évaluation des attributs qui y sont attachés sera précalculé et utilisé directement lors de l'évaluation.
3. **Méthodes à ordre fixe**
   on choisit un ordre d'évaluation sans tenir compte des règles sémantiques. Par exemple, si on couple l'évaluation avec l'analyse syntaxique, l'ordre d'évaluation sera imposé par la méthode d'analyse.
   Ex. LL à chaque fois qu'on expanse.
   LR à chaque qu'on réduit.

Méthode 1 : trop coûteuse car on calcule le graphe de chaque texte d'entrée.

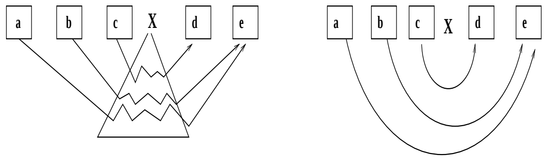
Méthode 3 : ex. YACC

Méthode 2 : système FNC-2.

Clairement, c'est la 2ème option qui accepte les spécifications les plus larges.

Comment, à partir des règles sémantiques, être sûr qu'il existe un ordre d'évaluation sans cycle et comment déterminer cet ordre ?

Graphe caractéristique inférieur :

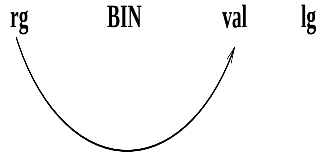


Soit t un arbre de la grammaire non contextuelle dont la racine est étiquetée par X ;
On restreint la fermeture transitive de $D(t)$ aux instances d'attributs de la racine, et on obtient une relation

$$D_{\uparrow t}(X) \subseteq Her(X) \times Syn(X)$$

dont le graphe est désigné par le graphe caractéristique inférieur pour X induit par t.

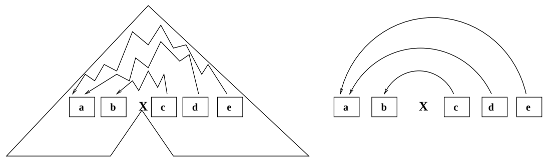
Exemple :



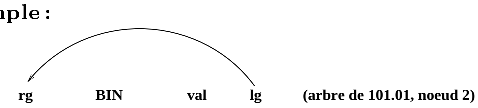
Graphe caractéristique supérieur :
Soit n un nœud interne de t étiqueté par X ; soit le sous-arbre t_n obtenu à partir de t en supprimant le sous-arbre de racine n, à l'exception du nœud n ;
On restreint la fermeture transitive de $D(t_n)$ aux instances d'attributs de n et on obtient une relation

$$D_{\downarrow t,n}(X) \subseteq Syn(X) \times Her(X)$$

dont le graphe est désigné par graphe caractéristique supérieur pour X induit par t en n.



Exemple :



On peut bien sûr ordonner arbitrairement les éléments comparables et transformer l'ordre partiel en ordre total.

En évaluant les instances d'attributs dans t selon l'ordre d'évaluation, les dépendances entre attributs seront respectées.

Soient t un arbre, n un nœud quelconque de cet arbre étiqueté X et p une production $X \to Y_1 \ldots Y_k$,
alors la relation de dépendance des instances d'attributs est déterminée par

$$D_p(p) \cup D_{\downarrow t,n}(X) \cup D_{\uparrow n_1}(Y_1) \cup \ldots \cup D_{\uparrow n_k}(Y_k)$$

Des arbres différents peuvent induire des relations $D_{\uparrow n_1}(Y_k)$ ou $D_{\downarrow t,n}(X)$ différents, mais seulement en nombre fini, puisqu'il y a un nombre fini et fixé d'attributs pour chaque symbole.

Si $|Her(X)| = n$ et $|Syn(X)| = m$, il existe au plus $2^{n.m}$ relations différentes.
Malheureusement, cette borne supérieure théorique détermine **effectivement** la complexité de quelques-uns des problèmes dans la génération des évaluateurs d'attributs.

On peut construire des graphes mais c'est coûteux.

Idée : on va approcher la dépendance globale en cumulant sur un même graphe toutes les dépendances issues de toutes les productions où un même symbole apparaît en partie gauche.

Vis-à-vis d'un arbre particulier, on aura trop de dépendances, i.e. en procédant ainsi, on peut trouver des cycles même s'il n'y en a pas en réalité.

En résumé, conceptuellement on peut diviser l'évaluation des attributs dans un arbre t en 2 phases :

1. la phase de stratégie : déterminer un ordre d'évaluation $T(t)$
2. la phase d'évaluation : qui effectue l'évaluation des attributs de t selon $T(t)$

Il existe des variantes pour chaque phase, (le choix d'une variante détermine une classe particulière d'évaluateurs).

I. La phase d'évaluation : elle peut commencer avec les éléments minimaux ou maximaux de $D(t)$.

Elément minimal : a est minimal par rapport à une relation R s'il n'existe aucun b tel que $b\ R\ a$
(a ne dépend d'aucun autre attribut)

Elément maximal : a est maximal par rapport à une relation R s'il n'existe aucun b tel que $a\ R\ b$
(aucun autre attribut ne dépend de a)

1. **Evaluation dirigée par la demande :** commence par les éléments maximaux. L'évaluateur est appelé avec un de ces éléments dont le calcul des valeurs nécessite récursivement l'évaluation des éléments suivants.
2. **Evaluation dirigée par les données :** commence par les éléments minimaux, i.e. les occurrences d'attributs qui ne dépendent d'aucune autre. On n'évalue l'élément suivant que si tous les arguments nécessaires à cette évaluation sont présents.

II. La phase de stratégie : comment déterminer un ordre d'évaluation pour l'arbre t ?

1. **Evaluation dynamique de l'ordre d'évaluation :** On effectue un tri topologique des graphes individuels $D(t)$ et on obtient un ordre total $T(t)$ sut $Oc(t)$.
2. **Ordre d'évaluation induit :** il existe une classe plus restreinte de grammaires attribuées, mais cependant suffisante pour les applications pratiques, *les grammaires l-ordonnées* qui ont les bonnes propriétés suivantes :

   pour tout non-terminal X, $\exists$ un ordre total T_X sur l'ensemble $Attr(X)$ des attributs de X tq, le graphe de dépendance de tout arbre soit compatible avec l'ordre total précalculé sur $Attr(X)$.

   En partant de ces ordres totaux, on peut former des ordres totaux sur les ensembles $Oc(p)$ d'occurrences d'attributs pour toutes les occurrences d'une production p quelconque (**séquence de visite**) qui, à leur tour, induisent un ordre d'évaluation unique $T(t)$ pour tout arbre t.

   Il n'y a plus qu'à suivre cet ordre pour évaluer.
3. **Sélection entre séquences de visite :** si une GA n'est pas l-ordonnée, $\exists$ des productions pour lesquelles on devra choisir des séquences de visite différentes selon le contexte.
   On peut calculer dès la génération toutes les séquences de visite pour chaque production. Pour un arbre donné, et pour toute application de production, le choix de la séquence correcte s'effectue dynamiquement (lors de l'évaluation).

On va caractériser différentes classes de GA pour pouvoir calculer ces ordres totaux entre attributs.

Remarque : 2 manières de coupler analyse syntaxique et évaluation des attributs :

1. analyse syntaxique $\Longrightarrow$ construction de l'arbre
   générateur d'évaluateur $\Longrightarrow$ génère des passes de visite de l'arbre avec évaluation des attributs.
   Avantage : il est possible d'évaluer en multi-passes.
2. évaluation des attributs au cours de l'analyse syntaxique.
   Avantage : on évalue les attributs en une passe (pas besoin de construire l'arbre syntaxique, du moins en ce qui concerne l'évaluation des attributs).
   Inconvénient : il y a des restrictions au niveau du pouvoir d'expression.

On s'intéresse à (2).

Evaluation des attributs dirigée par l'analyse syntaxique
Pour quelques classes de GA, restreintes mais cependant importantes en pratique à cause de leur faible consommation en temps et en mémoire.

L'évaluation des attributs a lieu en parallèle avec l'analyse syntaxique, et même, elle peut être guidée par cette analyse.

Les valeurs d'attributs sont toujours générées en pile. La construction de l'arbre syntaxique devient superflue, du moins en ce qui concerne l'évaluation des attributs.

1. Les grammaires S-attribuées Chaque règle sémantique n'utilise que des attributs synthétisés.

On va faire l'évaluation en parallèle avec l'analyse syntaxique.

L'analyseur est modifié de telle sorte qu'avant chaque réduction, il effectue le fragment de code correspondant aux règles sémantiques associées à la production par laquelle il réduit.

Exemple : soit la grammaire des expressions arithmétiques :

1.	E_0	$\rightarrow$	E_1 + T	{E_0.valeur := E_1.valeur + T.valeur}
2.	E	$\rightarrow$	T	{E.valeur := T.valeur}
3.	T_0	$\rightarrow$	T_1 * F	{T_0.valeur := T_1.valeur * F.valeur}
4.	T	$\rightarrow$	F	{T.valeur := F.valeur}
5.	F	$\rightarrow$	(E)	{F.valeur := E.valeur}
6.	F	$\rightarrow$	%*id*	{F.valeur := %*id*.vallex}
7.	F	$\rightarrow$	%*nb*	{F.valeur := %*nb*.vallex}

%*id*.vallex et %*nb*.vallex sont fournis par le scanner.

valeur : attribut synthétisé, de type int, associé à E, T, F

Analyse SLR(1) : fournit la table d'analyse suivante :

	%id	%nb	+	*	(	)	$	E	T	F
0	d5	d6			d4			1	2	3
1			d7				acc			
2			r2	d8		r2	r2			
3			r4	r4		r4	r4			
4	d5	d6			d4			9	2	3
5			r6	r6		r6	r6			
6			r7	r7		r7	r7			
7	d5	d6			d4				10	3
8	d5	d6			d4					11
9			d7			d12				
10			r1	d8		r1	r1			
11			r3	r3		r3	r3			
12			r5	r5		r5	r5			

chaîne d'entrée : $a * 2 + b\$$ avec a.vallex = 5, b.vallex = 3.

pile des états	pile des attributs	chaîne	action
0		$a * 2 + b\$$	décaler
0 a 5	5	$*2 + b\$$	r(F → %id)
0 F 3	5	$*2 + b\$$	r(T → F)
0 T 2	5	$*2 + b\$$	décaler
0 T 2 * 8	5	$2 + b\$$	décaler
0 T 2 * 8 2 6	5,2	$+b\$$	r(F → %nb)
0 T 2 * 8 F 11	5,2	$+b\$$	r(T → T*F)
0 T 2	10	$+b\$$	r(E → T)
0 E 1	10	$+b\$$	décaler
0 E 1 + 7	10	$b\$$	décaler
0 E 1 + 7 b 5	10,3	$	r(F → %id)
0 E 1 + 7 F 3	10,3	$	r(T → F)
0 E 1 + 7 T 10	10,3	$	r(E → E+T)
0 E 1	13	$	acc

C'est exactement le fonctionnement de Yacc + utilisation de variables globales.

2. Les grammaires H-attribuées définition symétrique à celle des S-attribuées. N'ont aucun rôle pratique.

3. Les grammaires L-attribuées

Une GA est L-attribuée ssi pour toute production $p : X_0 \to X_1 \dots X_{n_p}$ de la grammaire et pour toute règle sémantique $a_i = f_{p,a,i}(b^1_{j_1}, \dots, b^k_{j_k})$ de p, on a :

$$a \in Her(X_i) \text{ et } 1 \leq i \leq n_p \Rightarrow j_l < i, \forall l,\ 1 \leq l \leq k$$

i.e. les occurrences d'attributs hérités de $X_i,\ i \leq n_p$ ne dépendent que :

- des attributs de $X_1, \dots, X_{i-1}$
- des attributs hérités de X_0

Remarque : toute GA S-attribuée est aussi L-attribuée (puisque les contraintes ne portent que sur les attributs hérités).

Une GA L-attribuée n'a donc aucune dépendance de droite à gauche. Pour un tel type de grammaire, les attributs peuvent être évalués par l'algorithme suivant, (si l'arbre syntaxique est construit avant) :


```
Programme L-AA ;
procédure visite(n : nœud ; n.h : attributs hérités ; n.s : attributs synthétisés)
        % n.h : paramètres en entrée
        % n.s : paramètres en sortie
    cas prod(n) est                 p : X0 → X1 ... Xnp
          ...
          p : début
                  évaluer (Her(X1)) ; visite(n1, n1.h, n1.s) ;
                  évaluer (Her(X2)) ; visite(n2, n2.h, n2.s) ;
                  ...
                  évaluer (Her(Xnp)) ; visite(np, np.h, np.s) ;
                  évaluer (Syn(X0)) ;
          fin ;
    fincas ;
début
    évaluer (Her(ε)) ;          ε :racine de l'arbre
    visite(ε, ε.h, ε.s) ;
    écrire(ε.s)          résultat de l'analyse sémantique
fin.
```

Remarque : la grammaire Bin_to_Dec et la précédente (types) ne sont pas L-attribuées.

Grammaire LL-attribuée si

- la GA est L-attribuée
- et la grammaire sous-jacente est LL

Implantation d'une grammaire attribuée : lorsque l'analyseur LL entreprend l'expansion d'un non-terminal, tous les arguments de ses attributs hérités sont calculés, (ou peuvent être calculés).

PS : pile syntaxique. PA : pile des attributs.

Expansion d'un non-terminal B

PS avant: | $A \to \alpha.B\beta$ |

PS après: | $A \to \alpha.B\beta$ | $B \to .\gamma$ |

PA avant: | LHer(A) | LSyn(α) |

PA après: | LHer(A) | LSyn(α) | LHer(B) |

Lecture d'un symbole terminal a

PS avant: | $A \to \alpha.a\beta$ |

PS après: | $A \to \alpha a.\beta$ |

PA avant: | LHer(A) | LSyn(α) |

PA après: | LHer(A) | LSyn(α) | LSyn(a) |

Réduction selon B $\to \gamma$ (on a fini d'expanser B)

PS avant: $A \to \alpha.B\beta$ | $B \to \gamma.$ |

PS après: | $A \to \alpha B.\beta$ |

PA avant: LHer(A) | LSyn(α) | LHer(B) | LSyn(γ) |

PA après: | LHer(A) | LSyn(α) | LSyn(B) |

On peut facilement formuler un invariant sur le contenu de la pile des attributs par rapport au contenu de la pile syntaxique :

Si $A_1 \to \alpha_1.\beta_1$, $A_2 \to \alpha_2.\beta_2$, $\ldots$ $A_n \to \alpha_n.\beta_n$ est le contenu de PS, alors le contenu de PA est la liste des valeurs d'attributs pour

$$\text{LHer}(A_1)\ \text{LSyn}(\alpha_1)\ \text{LHer}(A_2)\ \text{LSyn}(\alpha_2) \ldots\ \text{LHer}(A_n)\ \text{LSyn}(\alpha_n)$$

4. Les grammaires LR-attribuées

Un analyseur LL sait, lors d'une expansion d'un non-terminal quelle est la production choisie, $\Longrightarrow$ aide pour localiser les attributs dans la pile.

L'analyseur LR, lui a en général plusieurs items dans son état qui décrivent les situations possibles de l'analyse. Le seul cas où on peut associer sans ambiguïté une production, c'est le cas où l'analyseur effectue une réduction.

Idée : lier toutes les actions d'évaluation des attributs aux réductions.

L'algorithme suivant peut prédire la position d'une valeur d'attribut dans la pile PA. De cette façon, l'évaluateur peut descendre dans PA pour y trouver cette valeur d'attribut :

Algorithme :
Pour toute production p : A $\rightarrow$ X$_1$ ... X$_n$, on introduit n nouveaux marqueurs M$_1$ ... M$_n$ et

- on remplace p par

$$\mathrm{A} \rightarrow \mathrm{M}_1\ \mathrm{X}_1 \dots \mathrm{M}_n\ \mathrm{X}_n$$

- on ajoute les règles $\mathrm{M}_j \rightarrow \epsilon,\ \forall j \in [1..n]$

Deux cas sont à envisager :

1. quand on réduit par $\mathrm{M}_j \rightarrow \epsilon$, on sait à quelle production $\mathrm{A} \rightarrow \mathrm{M}_1\ \mathrm{X}_1 \dots \mathrm{M}_n\ \mathrm{X}_n$ ce marqueur appartient. On connaît les positions de tous les attributs

   $$\mathrm{Her(A)|Her(X_1)|Syn(X_1)|Her(X_2)|Syn(X_2)|} \dots |(*)$$

   (*)On peut calculer Her(X$_j$) et le stocker au sommet de PA. C'est le nouveau sommet après réduction.

2. quand on réduit par $\mathrm{A} \rightarrow \mathrm{M}_1\ \mathrm{X}_1 \dots \mathrm{M}_n\ \mathrm{X}_n$, on doit calculer Syn(A). Les attributs intervenant dans ce calcul sont tous à des positions connues dans PA, celles des Syn(X$_j$) au moment de la réduction.

Remarque : Pour (1), il est important que la grammaire soit LL(1). (Elle l'est car LL(1) $\Rightarrow$ LR(1))
Conséquence : on est sûr de réduire ϵ vers un marqueur particulier (un symbole de prévision suffit).

Simplifications de l'algorithme : permettent de réduire le nombre de marqueurs à introduire dans une règle.

1. si X$_j$ n'a pas d'attribut hérité, M$_j$ est inutile.
   Bien sûr, il faudra décaler les positions des attributs dans PA.
2. si X$_1$.h existe, mais qu'il est calculé par une règle de copie X$_1$.h = A.h, on peut omettre M$_1$:
   au lieu d'utiliser X$_1$.h, on utilise A.h qui est juste au dessous dans PA.

Proposition : toute grammaire LL(1) reste LR(1) quand on lui ajoute des marqueurs.

Un attribut hérité t_c (taille de caractères) ; influe sur la hauteur d'une formule.

Le non-terminal B (pour boîte) représente une formule.

B $\rightarrow$ B B   réalise la juxtaposition de 2 boîtes.

B $\rightarrow$ B *sub* B   place la 2ème boîte en indice de la 1ère.

L'attribut *htn* de texte est obtenu par recherche dans une table indicée par le caractère représenté pour texte.

Rétrécir : réduit la taille de caractère de 30%.
Déplacer : prend en compte le déplacement vers le bas de la boîte pour calculer la hauteur.

En appliquant l'algorithme, la grammaire devient :

S $\rightarrow$	L B	{B.tc = L.s ; S.ht = B.ht}
L $\rightarrow$	ϵ	{L.s = 10}
B $\rightarrow$	B$_1$ M B$_2$	{B$_1$.tc = B.tc ;
		M.h = B.tc ;
		B$_2$.tc = M.s ;
		B.ht = max(B$_1$.ht, B$_2$.ht)} ;
M $\rightarrow$	ϵ	{M.s = M.h}
B $\rightarrow$	B$_1$ sub N B$_2$	{B$_1$.tc = B.tc ;
		N.h = B.tc ;
		B$_2$.tc = N.s ;
		B.ht = Déplacer(B$_1$.ht, B$_2$.ht)}
N $\rightarrow$	ϵ	{N.s = Rétrécir(N.h)}
B $\rightarrow$	texte	{B.ht = texte.htn × B.tc}

Conclusion : Vouloir faire l'évaluation des attributs en même temps que l'analyse syntaxique est très limitant.
Les systèmes performants actuels font des évaluations multi-passes sur l'arbre syntaxique déjà construit, les séquences de visite étant générées automatiquement et les tests de circularité étant effectués.

3. Traduction des langages impératifs

1. Les concepts des langages impératifs.
2. L'architecture de la machine virtuelle.
3. Affectations et expressions.
4. Les instructions.
   (a) Les instructions usuelles.
   (b) Les instructions conditionnelles.
   (c) Les instructions itératives.
5. Implantation mémoire des variables de type simple
6. Implantation des tableaux
   (a) Tableaux statiques
   (b) Tableaux dynamiques

7. Allocation de structures
8. Pointeurs et allocation dynamique
9. La TDS et les allocations
10. Procédures
    (a) Organisation mémoire
    (b) Adressage des variables
    (c) Appel et retour d'une procédure
       - Les instructions de gestion du bloc d'activation
    (d) Transmission des paramètres
    (e) Accès aux variables et paramètres formels
    (f) Procédures passées en paramètre
11. Le programme principal
12. Un exemple complet de traduction

C'est une description de ce que fait le compilateur et non de comment il le fait. On va décrire la correspondance

programme en langage source $\Longrightarrow$ programme obtenu par compilation
pour un calculateur cible

Pour illustrer, on choisit pour langage un Pascal *légèrement appauvri* et la machine cible est une machine SPARC simplifiée, mais de telle sorte que la distance à la machine réelle soit faible.

Pascal $\longrightarrow$ SPARC virtuel $\longrightarrow$ SPARC réel

3.1 Les concepts des langages impératifs

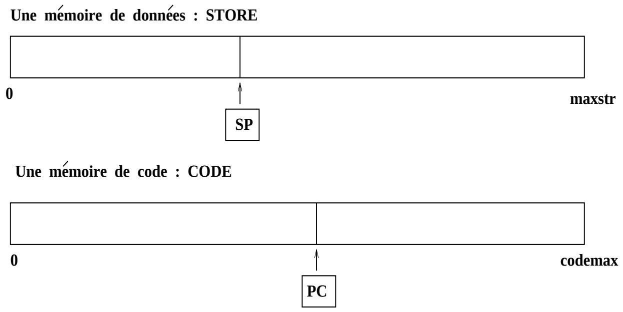
- **Les variables :** sont des *récipients* pour des objets dont le *contenu* (valeur) peut changer au cours de l'exécution du programme.
  Les valeurs des variables à un instant donné de l'exécution constituent l'état du programme à cet instant.

  $$\text{Un } \textit{nom} \text{ sert à désigner} \begin{cases} \text{une variable} \\ \text{une constante} \\ \text{une procédure} \end{cases}$$

  Les désignations de variables doivent être assignées à des cellules de la mémoire qui contiendront les valeurs courantes.
  L'appel d'une procédure récursive entraîne de nouvelles instanciations des désignations des variables locales.
  $\Longrightarrow$ leur réserver de nouvelles places en mémoire. A la fin de la procédure, les cellules utilisées pour ces instanciations sont libérées.
  Conséquence : ces langages doivent être implantés avec une gestion de la mémoire **en pile**.
- **Les expressions :** termes construits à partir de constantes, de noms et d'opérateurs, et évaluées lors de l'exécution. Leur valeur est généralement dépendante de l'état du programme.
- **la donnée explicite du flot de contrôle :**
  – branchement inconditionnel (goto, break)
  – branchement conditionnel (if, while, repeat, for)
  – case, réalisé de façon efficace par un branchement indexé.

3.2 L'architecture de la machine virtuelle

La mémoire et quelques registres :



L'architecture et les instructions sont introduites au fur et à mesure des besoins.

SP (Stack Pointer) pointe sur le sommet de pile (plus haute cellule occupée). A partir de l'adresse 0, la taille de la pile augmente et diminue successivement au cours de l'exécution.

Le registre PC est le pointeur d'instruction. Il contient l'adresse de la prochaine instruction à exécuter.

Toutes les instructions occupent **un mot mémoire**. Donc, en dehors des branchements, le contenu de PC est incrémenté de 1 à chaque fois. Le cycle de base est défini par :

```
do
  PC := PC + 1;
  executer CODE[PC - 1]
od;
```


Certaines instructions rangent le contenu d'adresse au sommet de pile (elles rallongent la pile). D'autres copient le contenu du sommet de pile à une adresse mémoire et racourcissent la pile.

On suppose qu'on a une infinité de registres entiers numérotés r_0, $r_1, \ldots, r_n, \ldots$

3.3 Affectations et expressions

Le SPARC est une architecture load-store : les instructions de calcul se font de regitres à registres et il n'y a que les *load* et les *store* qui font référence à la mémoire.

3.4 Les instructions
(a) Les instructions usuelles

Instructions arithmétiques entières

addition	$r_3 := r_1 + r_2$
soustraction	$r_3 := r_1 - r_2$
multiplication	$r_3 := r_1 * r_2$
division	$r_3 := r_1 / r_2$
negate	$r_2 := -r_1$

Instructions booléennes

et $r_3 := r_1$ and r_2

Le and se fait bit à bit sur tout le mot de 32 bits.

or et not sont définis de façon similaire.

La comparaison

cmp r_1, r_2 met à jour les codes condition (registres particuliers).

Les branchements

beq etiq teste si les codes condition signifient égalité.
Si oui, on se branche à etiq.

bne	etiq	branche si non égal
ble (blt)	etiq	branche si plus petit ou égal (si plus petit)
bge (bgt)	etiq	branche si plus grand ou égal (si plus grand)

etiq : étiquette remplacée à terme par l'adresse de l'instruction cible.

Chargement de constantes

mov r_1, const ou $r_1 :=$ const

si la constante n'est pas trop grande.

Instructions de lecture et écriture en mémoire

load : ld [r_1], r_2

le contenu de l'adresse qui est dans r_1 est chargé dans r_2

ld [r_1+const], r_2

store : st r_1, [r_2]

on charge la valeur de r_1 à l'adresse contenue dans r_2

st r_1, [r_2+ const]

Pour faire une indirection, si on veut charger la valeur dont l'adresse est la valeur stockée à l'adresse dont la valeur est dans r_1 :

ld [r_1], r_2
ld [r_2], r_3

Les affectations

- la désignation d'une variable en partie gauche représente l'adresse,
- celle en partie droite représente la valeur contenue dans l'adresse associée.

$$x := y \text{ se traduit par } \begin{cases} r_x := \text{ valeur gauche de } x \\ r_y := \text{ valeur droite de } y \\ \text{st } r_y,\ [r_x] \end{cases}$$

On a besoin de la fonction ρ qui assigne à toutes les variables déclarées, une adresse mémoire.
Pour l'instant, on suppose que c'est une adresse absolue :

$\rho\,:\,x \mapsto \rho(x)$ adresse dans STORE

$\rho(x)$ est une valeur stockée dans la TDS.

Exemple : un programme avec 3 variables entières $a,\ b,\ c$ qui ont pour adresses respectives 5, 6, 7. On traduit

A $= a := (b + (b * c))$

code(A) $=$ code$_G(a, \rho, r_1)$; code$_D(b + (b * c), \rho, r_2)$; st $r_2,\ [r_1]$

code$_G(a, \rho, r_1) = r_1 := 5$

code$_D(b + (b * c), \rho, r_2) =$
code$_D(b, \rho, r_3)$; code$_D(b * c, \rho, r_4)$; $r_2 := r_3 + r_4$

code$_D(b, \rho, r_3) = r_5 := 6$; ld$[r_5],\ r_3$

code$_D(b * c, \rho, r_4) =$
$r_6 := 6$; ld$[r_6],\ r_7$; $r_8 := 7$; ld$[r_8], r_9$; $r_4 := r_7 * r_9$

(b) Les instructions conditionnelles

On étiquette certaines instructions. L'instruction de branchement conditionnel qui aura une telle étiquette pour cible utilise en fait l'adresse de l'instruction qui porte cette étiquette.

ba q signifie PC := q condition : q $\in$ [0, codemax]

Branchement inconditionnel :
code(if e then st1 else st2, ρ) $=$
code$_D$(e, ρ) ; bcond L$_1$; code(st1,ρ) ; ba L$_2$; L$_1$: code(st2, ρ) ; L$_2$:

```
      codeD(e, ρ) ;
      bcond L1 ;
      code(st1, ρ) ;
      ba L2 ;
L1 :  code(st2, ρ) ;
L2 :
```

code(if e then st1, ρ) $=$

```
      codeD(e, ρ) ;
      bcond L1 ;
      code(st1, ρ) ;
L1 :
```

Remarque : ici, le branchement conditionnel teste *si faux*.
Souvent, sur une machine, c'est le contraire : on se branche si la condition est vraie. Dans ce cas, on traduit par :

```
     codeD(non(e), ρ) ;
     bcond L ;
     code(st1, ρ) ;
L :
```


(c) Les instructions itératives

code(while e do st, ρ) $=$

```
L1 :  codeD(e, ρ) ;
      bcond L2 ;
      code(st, ρ) ;
      ba L1
L2 :
```

code(repeat st until e, ρ) $=$

```
L1 :  code(st, ρ) ;
      codeD(e, ρ) ;
      bcond L1
```

Exemple : traduction de

while $a > b$ do c:= c+1 ;
if $c < b$ then $a := a - b$

avec $\rho(a) = 5$; $\rho(b) = 6$; $\rho(c) = 7$

Le code est le suivant :

%code($a > b,\ \rho$)%

L$_3$: r_1:=5; ld$[r_1],\ r_2$; $r_3 := 6$; ld$[r_3],\ r_4$; cmp $r_2,\ r_4$;

ble L$_1$;

%$c := c + 1$%

$r_5 := 7$; ld$[r_5],\ r_6$; $r_6 := r_6 + 1$; st $r_6,\ [r_5]$;

% if $c < b \ldots$%

$r_7 := 7$; ld $[r_7],\ r_8$; $r_9 := 6$; ld $[r_9],\ r_{10}$; cmp $r_8,\ r_{10}$;

bge L$_2$;

%$a := a - b$%

$r_{11} := 5$; ld $[r_{11}],\ r_{12}$; $r_{13} := 6$; ld $[r_{13}],\ r_{14}$;

$r_{15} := r_{12} - r_{14}$; st $r_{15},\ [r_{11}]$;

L_2 : ba L$_3$;

L$_1$:

Ce n'est pas du code très optimisé.

- ajouter 1 arrive très fréquemment. Il existe une instruction inc qui le fait de façon accélérée.

  bge L_2;

- ...

  L_2 : ba L_3;

  On remplace bge L_2 par bge L_3. Comme c'était la seule référence à L_2, on supprime L_2. Le code devient :

%code($a > b,\ \rho$)%

L_3 : r_1:=5; ld[r_1], r_2; r_3 := 6; ld[r_3], r_4; cmp r_2, r_4;

ble L_1 ;

%$c := c + 1$%

r_5 := 7; ld[r_5], r_6; inc r_6; st r_6, [r_5];

% if $c < b$...

r_7 := 7; ld [r_7], r_8; r_9 := 6; ld [r_9], r_{10}; cmp r_8, r_{10};

bge L_3;

%$a := a - b$%

r_{11} := 5; ld [r_{11}], r_{12}; r_{13} := 6; ld [r_{13}], r_{14};

$r_{15} := r_{12} - r_{14}$; st r_{15}, [r_{11}];

ba L_3;

L_1 :

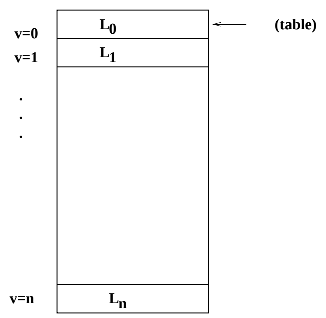
Traduction de
case e of
v_1 : st_1
v_2 : st_2
v_3 : st_3
autre : st_n
fin.

Dans le cas général, avec $v_1,\ v_2, \ldots, v_k$ ayant des valeurs quelconques et non 0, 1, ... deux façons de faire :

1) code pour évaluer e dans t
ba test
L_1 : code(st_1)
ba suite
L_2 : code(st_2)
ba suite
...
L_n : code(st_n)
ba suite
test : si t = v_1 aller à L_1
si t = v_2 aller à L_2
...
si t = v_{n-1} aller à L_{n-1}
aller à L_n
suite :

2) code pour évaluer e dans t
si t $\neq v_1$ aller à L_1
code(st_1)
ba suite
L_1 : si t $\neq v_2$ aller à L_2
code(st_2)
ba suite
L_2 : ...
L_{n-2} : si t $\neq v_{n-1}$ aller à L_{n-1}
code(st_{n-1})
ba suite
L_{n-1} : code(st_n)
suite :

Remarque : lorsque le nombre de cas dépasse 10, on fait des *tables d'aiguillage*.
Dans le cas où les valeurs des cas est la suite de valeurs 0, 1, ... n avec n pas trop grand, on peut utiliser la notion de branchement indexé.



Une indirection sur ρ(table) $+v$ donne l'adresse de l'instruction à exécuter.

code pour évaluer e dans t
y ajouter la valeur de ρ(table)
faire une indirection et se brancher à l'adresse indiquée.

3.5 Implantation mémoire des variables de type simple

Compilation : le programme P est traduit en code SPARC virtuel
exécution : P.code est exécuté.

Informations statiques : disponibles à la compilation (directement ou par calcul d'autres données statiques).

Informations dynamiques : ne sont connues qu'à l'exécution, (à partir du jeu de données).

Exemple : les adresses cibles des branchements sont calculées à la compilation : informations statiques.
Par contre, les valeurs des variables sont des informations dynamiques (en général). Par conséquent, les valeurs des conditions aussi, et donc le flot de contrôle.

Affectation des variables déclarées à des cellules en mémoire,
$\leadsto$ certaines hypothèses sur la taille des cellules.

A chaque variable de type simple (entier, réel, booléen) de type ensemble ou pointeur, de type énuméré, on associe une cellule pour stocker sa valeur.
(On ne se souciera pas de compacter dans une seule cellule plusieurs *petites* valeurs telles que des booléens, comme cela est fait en réalité, ou de prendre plus d'une cellule pour augmenter la précision des réels).

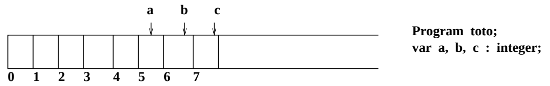
Dans un premier temps, on considère un programme Pascal sans procédure.

A chaque variable déclarée, on affecte une adresse dans la pile.
Affectation en séquence, dans l'ordre des déclarations.

Pour l'instant, on laisse libres les adresses de 0 à 4.

Les adresses sont dites relatives.

L'adresse absolue 5 est donc considérée comme une adresse relative par rapport à l'origine 0.



Les adresses relatives sont des données statiques. Elles sont calculées par un algorithme simple à partir du programme source. La fonction qui fixe l'affectation des variables à des adresses relatives est :
$\rho \; : \; Var \longrightarrow N_0$

3.6 Implantation des tableaux

D'abord, tableaux statiques (Pascal), puis tableaux dynamiques (Algol 60 ou ISO-Pascal).

(a) Tableaux statiques

var T : array[-5..5, 1..9] of integer ;

Combien de cellules occupe T ? (on suppose qu'un entier prend 1 octet de mémoire)
T contient (5-(-5)+1) * (9-1+1) éléments, donc T occupe 99 cellules.

On peut les ranger par lignes :

$$\underbrace{T[-5,1],\ T[-5,2],\ \ldots\ T[-5,9]}_{\text{1ère ligne}},\ \underbrace{T[-4,1],\ \ldots,\ T[-4,9]}_{\text{2ème ligne}},\ldots$$

Cas général à 2 dimensions : $T[l_1..u_1,\ l_2..u_2]$
adresse(T[i,j]) = $@(T[i,l_2]) + j - l_2$
$@(T[i,l_2])$=début de la $i^{\text{ème}}$ ligne

$$@(T[i,j]) = @(T[l_1,l_2]) + (i-l_1) * (u_2-l_2+1) + j-l_2$$
$$= @(T[l_1,l_2]) + i*(u_2-l_2+1) + j - (l_1*(u_2 - l_2+1) + l_2)$$

$$@(T[i,j]) = \underbrace{@(T[l_1,l_2]) - (l_1 * d_2 + l_2)}_{\text{partie constante}} + \underbrace{i*d_2 + j}_{\text{dépend des indices}}$$

avec $d_2 = u_2 - l_2 + 1$: capacité de la colonne.

Remarque : si $l_1 = l_2 = 0$, le calcul d'adresse devient plus simple comme en C :
$@(T[i,j]) = @(T[0,0]) + i*(u_2 +1) +j$
Dans le cas d'un tableau unidimensionnel :
$@(T[i]) = @(T[0]) + i$. C'est encore plus simple !

Généralisation : var T : array[$l_1..u_1, l_2..u_2, \ldots, l_k..u_k$] of integer ;

On pose $d_i = u_i - l_i +1$ = étendue du tableau dans la dimension i.
On calcule $@(T[i_1, i_2, \ldots, i_k])$ par rapport au début du tableau $@(T[l_1, l_2, \ldots, l_k])$

(i_1-l_1) * nbcell(array[$l_2..u_2, \ldots, l_k..u_k$] of integer) donne l'adresse du sous-tableau de dimension (k-1) où se trouve l'élément qu'on veut.

(i_1-l_1) * nbcell(array[$l_2..u_2, \ldots, l_k..u_k$] of integer) +
(i_2-l_2) * nbcell(array[$l_3..u_3, \ldots, l_k..u_k$] of integer) donne l'adresse du sous-tableau de dimension (k-2), ...

$@(T[i_1, i_2, \ldots, i_k])$ =
(i_1-l_1) * nbcell(array[$l_2..u_2, \ldots, l_k..u_k$] of integer) +
(i_2-l_2) * nbcell(array[$l_3..u_3, \ldots, l_k..u_k$] of integer) +
... +
$(i_{k-1}-l_{k-1})$ * nbcell(array[$l_k..u_k$] of integer) +
(i_k-l_k)

sachant que nbcell(array[$l_2..u_2, \ldots, l_k..u_k$] of integer = d_k

$$@(T[i_1, i_2, \ldots, i_k]) = (i_1-l_1) * d_2 * d_3 \ldots * d_k + (i_2-l_2) * d_3 * \ldots * d_k + \ldots + (i_{k-1}-l_{k-1}) * d_k + (i_k - l_k)$$

On isole les valeurs statiques (d_j, l_j) des valeurs dynamiques (i_j)

$$@(T[i_1, i_2, \ldots, i_k]) = (i_1*d_2*d_3*\ldots*d_k + i_2*d_3*\ldots*d_k + \ldots + i_{k-1}*d_k + i_k) - (l_1*d_2*d_3*\ldots*d_k + l_2*d_3*\ldots*d_k + \ldots + l_{k-1}*d_k + l_k)$$

Pour un calcul plus efficace :

@(T[i_1, i_2, …, i_k]) =

$$\underbrace{(((\ \ldots\ ((i_1 * d_2 + i_2) * d_3 + i_3) * d_4 + \ldots) * d_{k-1} + i_{k-1}) * d_k + i_k)}_{A}$$

$$- \underbrace{(((\ \ldots ((l_1 * d_2 + l_2) * d_3 + l_3) * d_4 + \ldots) * d_{k-1} + l_{k-1}) * d_k + l_k)}_{d}$$

Cas général :

var T :array[l_1..u_1,…,l_k..u_k] of t

où nbcell(t) = n (taille d'un objet de type t)

$$\boxed{@(T[i_1, i_2, \ldots, i_k]) = @(T[l_1,l_2,\ldots,l_k]) - d*n + A*n}$$

@(T[l_1,l_2,…,l_k])=$\rho(T)$: adresse du début du tableau

la valeur $\rho(T)$ - d*n : adresse de début fictive du tableau (connue à la compilation).

Remarque : pour faciliter le calcul de @(T[i_1, i_2, …, i_k]), on range dans la TDS, à l'entrée associée à T, les quantités nécessaires à ce calcul d'adresse, pour éviter de les recalculer chaque fois que nécessaire. On range aussi les l_i, u_i pour effectuer les tests de dépassement.

Exemple : var b : array[-5..5, 1..9] of integer ;

On suppose qu'un entier occupe 4 octets (n=4).

code$_G$(b[i,j], ρ, r_0) = $r_1 := \rho$(b) ; % $r_1 \leftarrow \rho$(b[-5,1]) %
code$_{Id}$([i,j] n, r_1, ρ, r_0)

code$_{Id}$([i,j] n, r_1, ρ, r_0) =
$r_1 := r_1$ + 176 ; %d*n = -44 * 4 = -176%
$r_2 := \rho$(i) ;
ld [r_2], r_3 ; %$r_3 \leftarrow$ i
$r_3 := r_3 * 9$; %d_2=9%
$r_4 := \rho$(j) ;
ld [r_4], r_4 ; %$r_4 \leftarrow$ j
$r_3 := r_3 + r_4$;
$r_3 := r_3 * 4$; %$r_3 \leftarrow (i * d_2 + j) * n$%
$r_0 := r_1 + r_3$;

(b) Tableaux dynamiques

La norme ISO de Pascal : on peut avoir des tableaux comme paramètres formels d'une procédure.

procédure p(T : array[l_1..u_1, …, l_k..u_k] of Type)

Les bornes de T ne sont définies qu'à l'appel de p.

Par ailleurs, l_i, u_i peuvent être des variables globales, ou des paramètres formels de la procédure. Alors T est un tableau dynamique.

Seules informations statiques : le nombre de dimensions (k) et la taille des éléments (puisqu'on n'a pas de polymorphisme).

Allocation statique des adresses pour un tableau dynamique impossible : la place nécessaire n'est connue qu'au moment de l'entrée dans la procédure. C'est à ce moment que la réservation en mémoire peut être effectuée.

L'allocation mémoire pour les tableaux dynamiques se fera dans une partie dynamique de la mémoire affectée à la procédure, derrière la partie statique.

Question : comment générer les instructions SPARC pour le calcul d'adresses sans connaître au moins l'adresse de début du tableau ?

On peut décider statiquement de l'endroit où déposer cette adresse de début lors de l'exécution : on utilise un **descripteur de tableau** dont l'adresse et la taille sont décidées statiquement.

Remarque : la taille du descripteur dépend uniquement du nombre de dimensions. On le schématise comme suit :

0	ρ(T)-d*n
1	d_1*d_2 … *d_k *n
2	d*n
3	n
4	l_1
5	u_1
	…
2i+2	l_i
2i+3	u_i
3k+1	d_2
	…
3k+($i-1$)	d_i

$\forall\ i$,	l_i à l'adresse 2*i+2
	u_i à l'adresse 2*i+3
	d_i à l'adresse 3*$k+(i-1)$

A l'entrée de la TDS associée au tableau, on range ρ(descripteur) et les valeurs de k et n.

Calcul d'adresse :

$\text{code}_G(\text{T}[\text{i}_1, \text{i}_2, \ldots, \text{i}_k], \rho, r_0])=$

$r_1 := \rho(\text{descripteur})$;

$\text{ld}[r_1], r'_1$; $\%r'_1 \leftarrow \rho(\text{T})$ -d*n%

$\text{code}_{Id}([\text{i}_1, \text{i}_2, \ldots, \text{i}_k]\text{n}, r'_1, \rho, r_0)$

$\text{code}_{Id}([\text{i}_1, \text{i}_2, \ldots, \text{i}_k]\text{n}, r'_1, \rho, r_0)=$

$r_2 := r_1 + 3k + 1$;

$\text{ld}[r_2], r_3$; $\%r_3 \leftarrow d_2\%$

$\text{code}_D(\text{i}_1, \rho, r_4)$; $\%r_4 \leftarrow i_1\%$

$r_5 := r_4 * r_3$;

$$\boxed{\begin{array}{l} \text{code}_D(\text{i}_2, \rho, r_4) ; \\ r_4 := r_5 + r_4 ; \\ r_2 := r_2 + 1 ; \\ \text{ld}[r_2], r_3 ; \\ r_5 := r_4 * r_3 ; \end{array}}$$

...

$\text{code}_D(\text{i}_{k-1}, \rho, r_4)$;

$r_4 := r_5 + r_4$;

$r_2 := r_2 + 1$;

$\text{ld}[r_2], r_3$; $\%r_3 \leftarrow \text{d}_k\%$

$r_5 := r_4 * r_3$; $\%r_5 \leftarrow (\ldots(\text{i}_1 * \text{d}_2 + \text{i}_2) * \text{d}_3 + \ldots) * \text{d}_{k-1} + \text{i}_{k-1}) * \text{d}_k$

$\text{code}_D(\text{i}_k, \rho, r_4)$;

$r_4 := r_5 + r_4$;

$r_0 := r'_1 + r_4$; $\%r_0 \leftarrow @(\text{T}[\text{i}_1, \text{i}_2, \ldots, \text{i}_k])$

Exemple : un tableau dynamique est déclaré dans une procédure :

var b : array[$\text{l}_1..\text{u}_1$, $\text{l}_2..\text{u}_2$] of integer ;

i, j : integer ;

Supposons

- à l'entrée de la procédure l_1=-4, u_1=5, l_2=1, u_2=10
- l'adresse relative de b est 5 (adresse du descripteur).
- le tableau est rangé dynamiquement à partir de l'adresse 20
- un entier occupe un octet.

$\text{code}_G(\text{b}[\text{i},\text{j}], \rho, r_0) = r_1 := 5 ; \text{ld}[r_1], r_0$

$r_2 := r_1 + 8 ; \text{ld}[r_2], r_3$;

$\%\text{code}_D(\text{i}, \rho, r_4)\%$

$r_4 := 14 ; \text{ld}[r_4], r_4$;

$r_5 := r_4 * r_3$; $\%r_5 \leftarrow$ i*10%

$\%\text{code}_D(\text{j}, \rho, r_4)\%$

$r_4 := 15 ; \text{ld}[r_4], r_4$;

$r_4 := r_5 + r_4$;

$r_0 := r_0 + r_4$;

5	20+39=59
6	100
7	-39
8	1
9	-4
10	5
11	1
12	10
13	d_2=10
ρ(i) = 14	
ρ(j) = 15	

3.7 Allocation des structures

Pour simplifier : cas des structures sans variantes, avec des types pour les composantes complètement expansés.

var v : record c_1 : t_1 ; c_2 : t_2 ; ... c_k : t_k end

On obtient l'adresse relative du début de la structure :

$$\rho(c_i) = \sum_{j=1}^{i-1} nbcell(t_j) \text{ et}$$

$$nbcell(t) = \sum_{i=1}^{k} nbcell(t_i).$$

En Pascal, nbcell(t) est statique $\Rightarrow$ connue à la compilation.
L'adressage d'une composante dans la structure se fait

1. en chargeant l'adresse de début de la structure ;
2. en ajoutant à cette adresse, l'adresse relative de la composante.

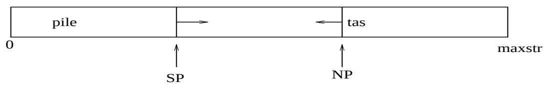
On traduit $\text{code}_G(v.c_i, \rho, r_0)$ par

$$\boxed{\begin{array}{l} r_1 := \rho(v) ; \\ r_2 := \rho(c_i) ; \\ r_0 := r_1 + r_2 ; \end{array}}$$

3.8 Pointeurs et allocation dynamique

- lien très fort entre pointeurs et allocation d'objets dynamiques.
- les objets pointés ne sont pas créés à la suite d'une déclaration, mais par une instruction appropriée (*new* de Pascal).
- En Pascal, en C, la sémantique relative à la durée de vie d'un objet créé dynamiquement n'est pas très précise.
- l'implantation du langage peut être telle que, si un objet devient inaccessible, alors on peut libérer la zone mémoire occupée $\leadsto$ ramasse-miettes (*garbage collection*)

Jusqu'à présent, tout fonctionnait avec une pile (ne convient plus).
On alloue de la place mémoire à ces objets dans la partie supérieure de la mémoire (*tas* ou *heap*).



La création d'un nouvel objet se fera au moyen de l'instruction *malloc*.

On doit avoir :
- la taille de l'objet à créer ;
- l'adresse de la variable qui contiendra le pointeur vers le futur objet. Cette adresse permettra de trouver indirectement l'adresse du nouvel objet.

Soit p un pointeur, l'allocation $new(p)$ se traduit par :

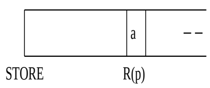
$r_0 :=$ taille de $p\uparrow$;	
$r_1 := malloc(r_0)$;	alloue dans le tas r_0 cellules pour ranger $p\uparrow$ on récupère dans r_1 l'adresse de cette allocation
$r_2 := \rho(p)$;	
st $r_1,\ [r_2]$;	

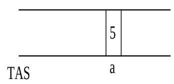
Remarque : Cette allocation doit, pour être tout à fait correcte, inclure le test que le tas et la pile ne se recouvrent pas. On utilise pour cela le registre EP (*extrêmité de la pile*).
EP indique à chaque instant, la plus haute adresse vers laquelle SP est susceptible de pointer lors de l'évaluation des expressions.
(Pour chaque expression, il est possible de calculer à l'avance le nombre maximal de cellules nécessaires pour l'évaluer) (à voir plus tard).

Traduction de $p\uparrow := 5$, p pointeur sur un entier.

$$\begin{aligned}\text{code}(p\uparrow := 5, \rho) = {} & \text{code}_G(p\uparrow, \rho, r_0)\,; \\ & r_1 := 5\,; \\ & \text{st } r_1,\ [r_0] \qquad \% \ r_0 \leftarrow a\ \%\end{aligned}$$

$$\begin{aligned}\text{code}_G(p\uparrow, \rho, r_0) = {} & r_2 := \rho(p)\,; \\ & \text{ld } [r_2],\ r_0\end{aligned}$$



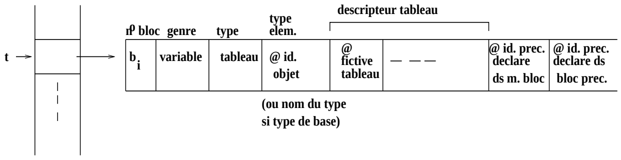


Conclusion : fonctions de codage supplémentaires pour des expressions arbitrairement complexes.

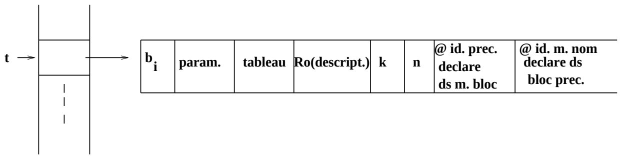
- $\text{code}(x\ r,\ \rho) = \text{code}_G(x\ r,\ \rho,\ r_0)$
- $$\text{code}_G(x\ r,\ \rho,\ r_i) = \left|\begin{array}{l} r_i := \rho(x) \\ \text{code}_M(r,\ \rho,\ r_i)\end{array}\right.$$
- $$\text{code}_M(.x\ r,\ \rho,\ r_i) = \left|\begin{array}{l} r_j := r_i + \rho(x) \\ \text{code}_M(r,\ \rho,\ r_j)\end{array}\right.$$
- $$\text{code}_M(\uparrow r,\ \rho,\ r_i) = \left|\begin{array}{l} \text{ld}[r_i],\ r_j \\ \text{code}_M(r,\ \rho,\ r_j)\end{array}\right.$$
- $$\text{code}_M([i]\ r, \rho,\ r_i) = \left|\begin{array}{l} \text{code}_{Id}([i]\ nb, r_i,\ \rho,\ r_j) \\ \text{code}_M(r,\ \rho,\ r_j)\end{array}\right.$$
- $\text{code}_M(\epsilon,\ \rho,\ r_i) = r_0 := r_i$

3.9 La TDS et les allocations

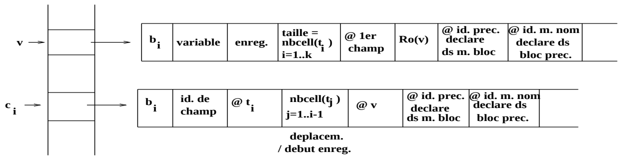
- var t : array[-5..5, 1..9, ...] of objet ;



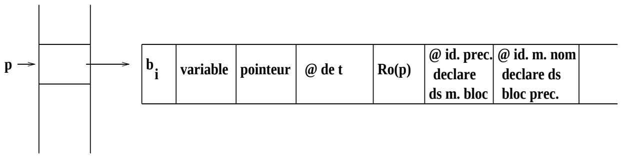
- procédure p(t : array[l_1..u_1, l_2..u_2, ..., l_k..u_k] of objet) ;



- var v : record c_1 : t_1 ; c_2 : t_2 ; ... c_k : t_k end ;



- var p : $\uparrow t$;



Exemple : Soient les déclarations suivantes :

$$
\begin{array}{lll}
\text{type } t = & \text{record} & a \ : \ \text{array}[-5..5, 1..9] \text{ of integer ;} \\
 & & b \ : \uparrow t \\
 & \text{end ;} & \\
\text{var} & i,\ j \ : & integer\,; \\
 & pt \ : & \uparrow t\,;
\end{array}
$$

Supposons que $\rho(i) = 5$ et qu'un entier occupe 4 octets,
Alors, $\rho(j) = 9,\ \rho(pt) = 13$.
On traduit $pt \uparrow .b \uparrow .a[i+1, j]$ par :

$$\text{code}(pt \uparrow .b \uparrow .a[i+1,j], \rho) = \text{code}_G(pt \uparrow .b \uparrow .a[i+1,j], \rho, r_0)$$

$$\text{code}_G(pt \uparrow .b \uparrow .a[i+1,j], \rho, r_0) = \left| \begin{array}{l} r_0 := 13\,; \\ \text{code}_M(\uparrow .b \uparrow .a[i+1,j], \rho, r_0) \end{array} \right.$$

$$\text{code}_M(\uparrow .b \uparrow .a[i+1,j], \rho, r_0) = \left| \begin{array}{l} \text{ld}[r_0],\ r_1\,; \\ \text{code}_M(.b \uparrow .a[i+1,j], \rho, r_1) \end{array} \right.$$

$$\text{code}_M(.b \uparrow .a[i+1,j], \rho, r_1) = \left| \begin{array}{l} r_2 := r_1 + 396\,; \\ \text{code}_M(\uparrow .a[i+1,j], \rho, r_2) \end{array} \right.$$

$$\text{code}_M(\uparrow .a[i+1,j], \rho, r_2) = \left| \begin{array}{l} \text{ld}[r_2],\ r_3\,; \\ \text{code}_M(.a[i+1,j], \rho, r_3) \end{array} \right.$$

$$\text{code}_M(.a[i+1,j], \rho, r_3) = \left| \begin{array}{l} r_3 := r_3 + 0\,; \\ \text{code}_M([i+1,j], \rho, r_3) \end{array} \right.$$

$$\text{code}_M([i+1,j], \rho, r_3) = \left| \begin{array}{l} \text{code}_{Id}([i+1,j]4, r_3, \rho, r_4) \\ \text{code}_M(\epsilon, \rho, r_4) \end{array} \right.$$

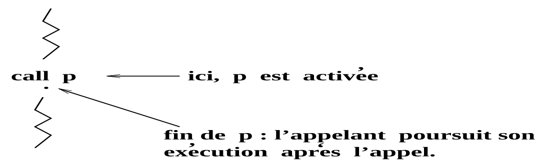
$$\text{code}_{Id}([i+1,j]4, r_3, \rho, r_4) = \left| \begin{array}{l} r_3 := r_3 + 176\,; \\ r_6 := 5\,; \\ \text{ld}[r_6],\ r_7\,; \\ r_7 := r_7 + 1\,; \\ r_8 := r_7 * 9\,; \\ r_9 := 9\,; \\ \text{ld}[r_9],\ r_{10}\,; \\ r_{11} := r_8 + r_{10}\,; \\ r_{12} := r_{11} * 4\,; \\ r_4 := r_3 + r_{12}\,; \end{array} \right.$$

$$\text{code}_M(\epsilon, \rho, r_4) = r_0 := r_4\,;$$

3.10 Procédures

La déclaration d'une procédure comprend :

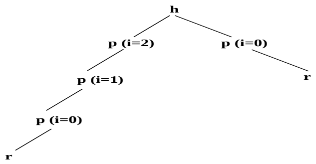
- un nom sous lequel elle peut être appelée ;
- la spécification des paramètres formels (paramètres d'entrée, paramètres de sortie) ;
- une suite de déclarations locales ;
- le corps de la procédure : une suite d'instructions ;
- le type du résultat lorsque la procédure est une fonction.



L'enchaînement des appels de procédure pendant l'exécution d'un programme forme un arbre : **l'arbre des appels**. La racine de cet arbre est étiquetée par le programme principal.

Chaque apparition d'un nom de procédure p est une instance de p caractérisée par le chemin depuis la racine jusqu'à ce nœud (*chemin d'instanciation*).

```
program h
  var i : integer ;
  proc p
    proc r
      ...
    if i > 0
    then i := i − 1 ; p
    else r
    fi ;
  i := 2 ;
  p ;
  p
end.
```



∀ nœud intérieur dans l'arbre, étiqueté par p :

- le nœud père est étiqueté par l'appelant ;
- ses nœuds fils correspondent aux procédures appelées depuis p, dans l'ordre de leur appel ;
- toutes les instances précédant p ont déjà été appelées mais pas encore quittées : *instances vivantes* à cet instant.

Remarque : à un programme, peuvent correspondre plusieurs arbres d'appel. Un arbre d'appel peut être infini.

Noms locaux : noms de paramètres et noms introduits par des déclarations locales. Par opposition aux **noms globaux**.

A l'appel d'une procédure, on crée de nouvelles instances pour les variables locales ⇒ réservation de places conformément aux spécifications ou déclarations pour les variables locales et conformément aux paramètres effectifs pour les paramètres.

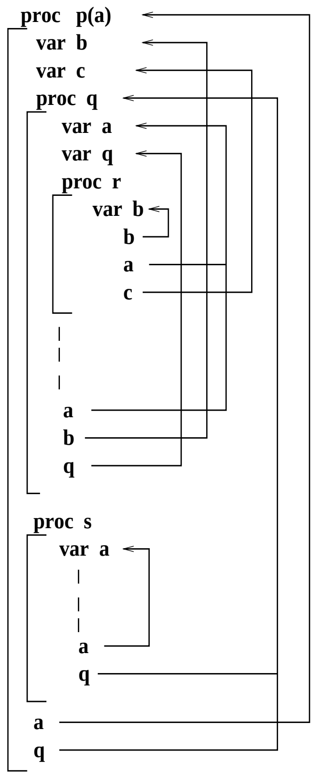
La durée de vie de ces instances = durée de la procédure ⇒ on libère la place en quittant la procédure (gestion de la mémoire par une pile).

Pour les variables globales, il faut tenir compte des règles de visibilité.

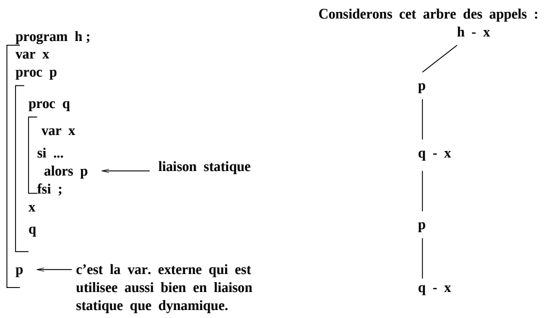
Lien statique : A chaque utilisation d'un nom global (à l'exécution) intervient une instance de la définition qui lui est liée statiquement. Ce lien ne repose que sur le texte du programme.

Lien dynamique : Un accès à un nom global fait intervenir la dernière instance de ce nom, quelle que soit la procédure dans laquelle on se trouve.

Exemple 1 : on montre pour chaque utilisation de variable, la définition associée.



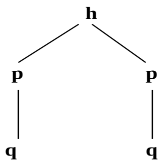
Exemple 2 :



- **liaison dynamique :** on remonte jusqu'à la 1ère instance de procédure contenant une déclaration de x.
- **liaison statique :** on cherche sur le chemin d'instanciation la procédure englobante la plus interne contenant une déclaration de x. Ici, c'est h.

Arbre des prédécesseurs statiques : pour rechercher de façon efficace l'instance correcte d'une variable globale.

pour l'arbre des appels ci-dessus, on a l'arbre des prédécesseurs statiques suivant : (un nœud est une instance de procédure et son père est la dernière instance créée de procédure englobante).



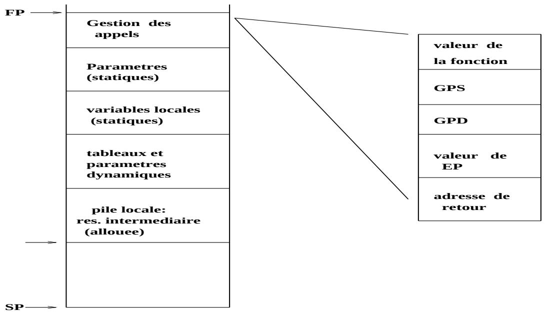
3.10.1 Organisation mémoire pour les procédures

- A un instant donné, toutes les instances vivantes de procédures (y compris le programme principal) constituent le chemin d'instanciation courant.
- En quittant une procédure, on remonte d'un niveau dans l'arbre des appels récursifs vers son père.
- Les variables accessibles à partir de l'instance courante se trouvent dans l'arbre des prédécesseurs statiques associé au chemin d'instanciation courant.

Dans l'organisation de la mémoire suivante (pile d'exécution),

- on réserve de la place pour les instances courantes, dans l'ordre de leur apparition sur le chemin d'instanciation.
- on lie entre eux les prédécesseurs statiques pour accéder efficacement aux variables globales (de l'intérieur vers l'extérieur).
- on lie entre eux les prédécesseurs dynamiques pour faciliter le retour de procédure.
- des cellules supplémentaires sont ajoutées pour gérer l'appel.

Pour chaque instance de procédure, une zone mémoire est réservée : c'est le bloc d'activation de la procédure (*stack frame*) avec la structure suivante :



On fait l'hypothèse qu'on sait calculer la hauteur maxi EP pour une procédure dans la gestion du calcul des expressions.

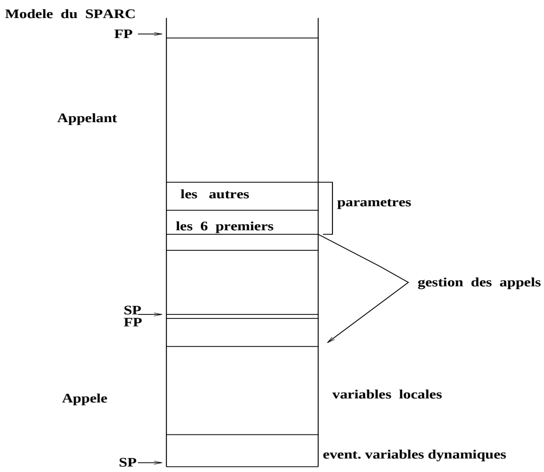
Les 5 premières cellules fixent la gestion des appels

1. pour contenir la valeur de retour dans le cas d'une fonction
2. pointeur sur le bloc d'activation du prédecesseur statique (pour remonter aux variables globales)
3. pointeur sur le bloc d'activation du prédecesseur dynamique : celui qui a appelé cette procédure (pour restaurer l'environnement de l'appelant au retour)
4. la valeur de EP
5. l'adresse de retour, i.e. l'instruction à exécuter au retour de la procédure.

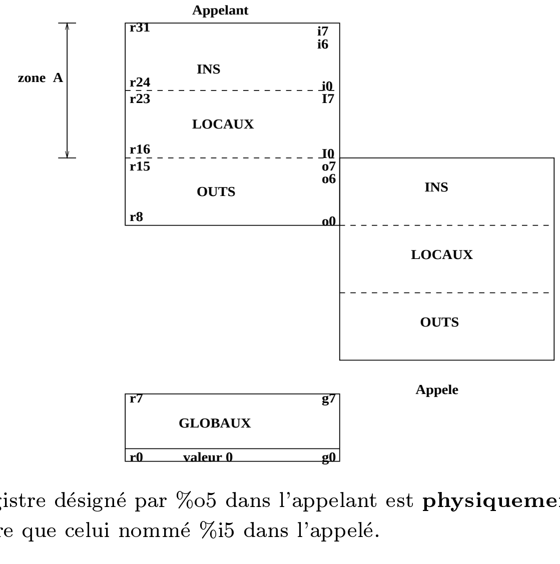
Ceci est un modèle qui correspond à une représentation sur une machine traditionnelle qui

- n'a pas de fenêtres de registres
- ou bien a peu de registres et ne permet pas *d'institutionnaliser* le passage de paramètres par des registres dédiés.

Dans le cas du SPARC, on a la notion de fenêtres de registres :



La gestion des appels se fait par le biais de 32 registres dédiés.



Le registre désigné par %o5 dans l'appelant est **physiquement** le même registre que celui nommé %i5 dans l'appelé.

La zone A de 16 mots mémoire est prévue pour sauver les ins et les locaux de l'appelant avant de faire l'appel à l'appelé (16 mots mémoire pour sauver 2*8 registres de 1 mot)

3.10.2 Adressage des variables

Procédures avec des règles de visibilité des variables $\Longrightarrow$ une variable ne peut pas avoir une adresse absolue dans STORE. Il faut associer à chaque variable son adresse relative par rapport au début du bloc d'activation où elle est définie.

L'accès à une variable se fait désormais :

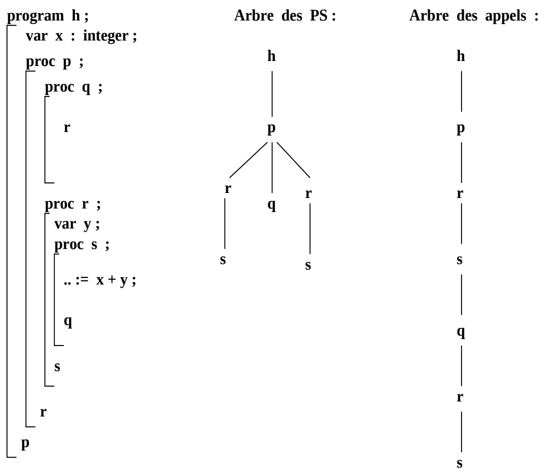
1. dans le cas d'une variable locale :
   - on accède à l'adresse du début du bloc d'activation qui est dans le registre FP
   - on ajoute l'adresse relative de la variable.
2. dans le cas d'une variable globale :
   Un accès depuis une profondeur n à une variable définie à une profondeur m avec bien sûr $m \leq n$ nécessite de suivre $n - m$ fois le pointeur PPS : on aura alors l'adresse du début du bloc d'activation où se trouve la variable. Il suffit d'ajouter l'adresse relative de cette variable.
   Remarque : $n = m$: cas des variables locales.

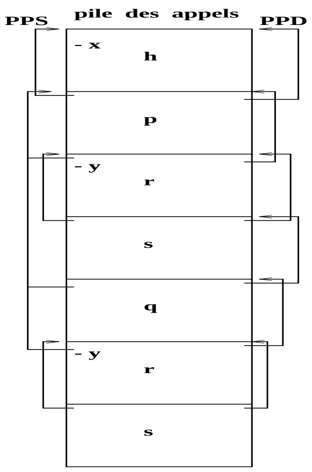
Pour pouvoir accéder à une variable globale, il faut mémoriser son adresse relative mais aussi avoir le moyen de calculer sa différence d'imbrication.

Si x est une variable, $\rho(x) =$ (@ relative, prof. d'imbrication)

La profondeur d'imbrication est définie par :

- celle du PP = 0
- la prof. d'imbrication d'une utilisation ou d'une définition dans une procédure de prof. n est $n + 1$.





définition	utilisation
p=1	p =1
q=2	q=4 (utilisé dans s de prof. 3)
x=1	x=4
r=2	r=2 (dans p)
	r=3 (dans q)
y=3	y=4
s=3	s=3 (dans r)

Pour atteindre la définition de x, depuis s, il faut suivre 3 fois le pointeur statique.

Adressage des paramètres passés par valeur : on peut les considérer comme des variables locales. Ils sont adressés au moyen de leur adresse relative par rapport au sommet de pile.

Adressage des paramètres passés par référence : on stocke leur adresse dans le bloc d'activation. On ne réserve donc qu'une cellule qui contient leur adresse. On accèdera toujours à ces paramètres indirectement.

p désigne la différence de profondeur d'imbrication (utilisation - définition) et q l'adresse relative :

- lireval(q, p), r_y, r_x
  si p=0 alors ld[%r_y+q], r_x
  sinon ld[%r_y+1], r_z
  lireval(q, p-1), r_z, r_x
  fsi;
- lireadr(q, p), r_y, r_x
  si p=0 alors r_x := %r_y+q
  sinon ld[%r_y+1], r_z
  lireadr(q, p-1), r_z, r_x
  fsi;

Calcul de l'environnement : pour un nom de procédure, ρ doit donner :

- le nom de l'étiquette (sa *valeur* est l'adresse de début du code de la procédure).
  Indispensable pour produire le branchement vers cette procédure.
- sa profondeur d'imbrication.

$$\text{Env} : \text{Nom} \rightarrow \text{Adr} \times \text{PI}$$

PI = ens. des entiers naturels.
Adr = ensemble des adresses de STORE (si nom de variable, @. relative ds STORE) et de CODE (si nom de procédure).

Il faut assurer qu'en tout point où ρ s'applique à un nom, ρ ne connaît que l'@. relative de l'occurrence de définition de ce nom normalement visible en ce point.

Pour chaque procédure, lors du traitement de la partie déclarative de cette procédure, on réécrit les entrées de l'environnement ρ hérités de l'extérieur pour les noms recouverts par une nouvelle déclaration.

Exemple : On suppose x, y, z connus avant de traiter la partie déclarative de p, avec ρ(x) = (12, 2), ρ(y) = (18, 2), ρ(z) = (16, 1).
On traite la partie déclarative de p, avec p de prof. d'imbrication 5 :
procédure p(params)
var x
ρ devient : ρ(x) = (2, 6), ρ(y) = (18, 2), ρ(z) = (16, 1).

Il faut donc, à la compilation, gérer une TDS qui permette de tenir compte des règles de visibilité des variables.
$\rightsquigarrow$ structure vue précédemment.

On définit les fonctions qui effectuent le calcul d'adresses :

Calculer $code(procedure\ p(specs)\,;vdecls\,;pdecls\,;corps,\ \rho,\ p_i)$ nécessite la définition de 3 fonctions de calcul d'environnement, une pour les paramètres, une pour les var. locales, et la troisième pour les procédures.

1. $elab\_spec\ :\ Spec^* \times Env \times Adr \times PI \rightarrow Env \times Adr$
$elab\_spec(list\_param,\ \rho,\ A,\ pi) = (\rho',\ B)$
où

- $list\_param$ = liste de paramètres
- A = valeur de la prochaine adresse relative à attribuer,
- pi = prof. d'imbrication de la procédure.

elab_spec retourne la nouvelle fonction d'environnement ρ' et la prochaine @. à attribuer B, après avoir attribué des @. aux paramètres.

a) Pour un paramètre passé par référence, on stocke une adresse :
$elab\_spec(var\ x\ :\ t\,;list\_param,\ \rho,\ A,\ pi) =$
$\quad elab\_spec(list\_param,$
$\qquad entrer\_id(\rho,\ x,\ A,\ param\_par\_ref),\ A+1,\ pi)$

entrer_id est la fonction d'entrée d'un identificateur dans la TDS. On stocke aussi les info. supplémentaires comme l'@. relative et le type de paramètre. Cette fonction modifie bien sûr la TDS et donc ρ.

b) pour un paramètre passé par valeur :
$elab\_spec(x\ :\ t\,;list\_param,\ \rho,\ A,\ pi) =$
$\quad elab\_spec(list\_param,$
$\qquad entrer\_id(\rho,\ x,\ A,\ param\_par\_val),\ A+nbcell(t),\ pi)$
où t est un type statique.

c) pour un tableau dynamique passé par valeur, on stocke le descripteur, i.e. pour un tableau de dimension k, $3k+3$ cellules :
$elab\_spec(x\ :\ array[l_1..u_1,\ldots,l_k..u_k]\ of\ t\,;list\_param,\ \rho,\ A,\ pi) =$
$\quad elab\_spec(list\_param,$
$\qquad entrer\_id\_tab(\rho,\ x, l_1, u_1, \ldots, l_k, u_k, d_2, \ldots, d_k,\ A,\ tab\_dyn),$
$\qquad A+3k+3,\ pi)$
où *entrer_id_tab* mémorise l'identificateur x mais aussi les données du descripteur $(l_1, u_1, \ldots, d_2, \ldots, d_k)$.

d) $elab\_spec((),\ \rho,\ A,\ pi) = (\rho,\ A)$

2. On définit *elab_vdecl* qui se charge de la réservation en mémoire des variables.
Dans le cas d'une variable de type simple, ou d'un tableau statique :

$$
\begin{aligned}
&elab\_vdecl(var\ x\ :\ t\,;\, vdecls,\ \rho,\ A,\ pi) = \\
&\quad elab\_vdecl(vdecls, \\
&\qquad entrer\_id(\rho,\ x,\ A,\ var),\ A + nbcell(t),\ pi)
\end{aligned}
$$

3. La fonction *elab_pdecl* traite les déclarations de procédures.

$$elab\_pdecl\ :\ Pdecl^{*} \times Env \times PI \rightarrow Env \times CODE$$

$$
\begin{aligned}
&elab\_pdecl(proc\ p_1(\ldots)\ suite\_p_1\,; \\
&\qquad \ldots \\
&\qquad proc\ p_k(\ldots)\ suite\_p_k\,;)\ \rho\ pi = \\
&\quad (\rho',\ L_1\ :\ code(procedure\ p_1(\ldots)\ suite\_p_1,\ \rho',\ pi+1) \\
&\qquad L_2\ :\ code(procedure\ p_2(\ldots)\ suite\_p_2,\ \rho',\ pi+1) \\
&\qquad \ldots \\
&\qquad L_k\ :\ code(procedure\ p_k(\ldots)\ suite\_p_k,\ \rho',\ pi+1) \\
&\quad) \\
&\text{avec } \rho' = entrer\_id(entrer\_id(\ldots(entrer\_id(\rho,\ p_1,\ L_1,\ proc), \\
&\qquad p_2,\ L_2,\ proc)\ldots,\ p_k,\ L_k,\ proc)
\end{aligned}
$$

entrer_id est appelé avec le niveau d'imbrication de la procédure englobante, i.e. *pi*.
elab_pdecl change l'environnement puisqu'elle associe à chaque nom de procédure une étiquette qui représente l'adresse de la première instruction de cette procédure. Elle produit aussi le code associé à ces procédures.

3.9.4 Appel et retour d'une procédure

Compiler une procédure, c'est savoir comment

- faire l'appel :
  - sauvegarder l'environnement de l'appelant,
  - transmettre les paramètres,
  - calculer et installer l'environnement de l'appelé,
  - passer à l'exécution du corps de l'appelé
- désigner les variables locales et les paramètres de l'appelé,
- exécuter le corps de l'appelé,
- terminer la procédure de l'appelé :
  - revenir à l'instruction suivant l'appel dans l'appelant,
  - rétablir l'environnement de l'appelant.

Soit p la procédure courante, PPS_p et PPD_p les pointeurs sur les prédecesseurs statique et dynamique de p ;
Soit q la procédure appelée par p, PPS_q et PPD_q les pointeurs sur les prédecesseurs statique et dynamique de q ;
Les tâches à effectuer avant d'exécuter q consistent à créer et remplir son bloc d'activation. Plus précisément :

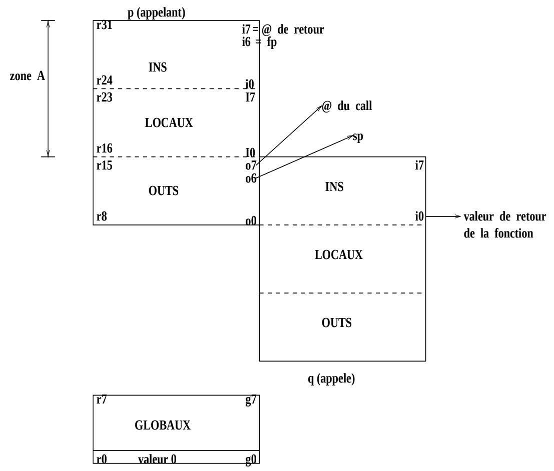
1. sauvegarder l'environnement de calcul de p, i.e. la valeur de certains registres (on peut les sauvegarder ou faire en sorte que q ne les écrase pas) ;
2. calculer PP_q : il doit pointer sur le début du bloc d'activation de la bonne instance de la procédure englobant immédiatement q sur le chemin d'instanciation. Il faut ranger cette valeur dans le bloc de q ;
3. calculer PPD_q (PPD_q = début du bloc d'activation de p)
4. le registre FP doit être mis à jour pour pointer sur le nouveau bloc.
   $FP := SP$ la nouvelle valeur de FP est égale à l'ancienne valeur de SP.
   Remarque : pas besoin de sauvegarder l'ancien FP car c'est la valeur de PPD_q
5. l'adresse de retour de l'appelant doit être sauvée dans le bloc d'activation de q
6. SP doit être mis à jour pour pointer sur le haut du bloc d'activation de q
7. les valeurs ou adresses des paramètres effectifs doivent être évaluées et écrites en mémoire. En particulier, il faut allouer des descripteurs de tableaux dynamiques.
8. les copies des valeurs des paramètres tableaux passés par valeur doivent être faites. S'il s'agit d'uyn tableau dynamique, il faut aussi allouer la place nécessaire.
9. faire le branchement vers la première instruction de q

Pour terminer q, il faut :
10. rétablir les anciennes valeurs de SP et FP
11. donner à PC l'adresse de l'instruction qui suit l'appel à q dans p
12. rétablir les valeurs des registres sauvegardés avant l'appel.

Les tâches 7 et 9 sont toujours à la charge de l'appelant. Les tâches 10 et 11 sont toujours à la charge de l'appelé.
Les autres tâches sont effectuées soit par l'appelant soit par l'appelé. Ça dépend des instructions de la machine. Dans le cas du SPARC, c'est presque toujours l'appelé qui effectue les reste des tâches.

(a) Les instructions de gestion du bloc d'activation :

- call etiq, n
  - effectue le branchement à l'étiquette etiq, i.e. la tâche 9 (n est le nombre de paramètres passés)
  - la valeur de PC qui est à l'adresse du call est écrite dans %o7 de l'appelant, i.e. %i7 quand la fenêtre de registres aura été incrémentée de 1 : c'est la tâche 5.



- save %sp, valimm, %sp
  - incrémente la fenêtre de registres de 1, i.e. qu'on a sauvegardé les INS et LOCAUX de l'appelant. En particulier, r30 qui contient %fp de l'appelant, on pourra donc le restaurer. (On n'est pas sûr de pouvoir le lire depuis q puisque la partie réservée à la sauvegarde de la fenêtre de registres est entièrement gérée par le processeur).
  - en incrémentant la fenêtre de registres de 1, %sp de p qui était dans %o6 est maintenant dans %i6, i.e. à la place de %fp de q et l'@ de call est dans %i7 (fin de la tâche 5).
  - valimm est la valeur dont il faut incrémenter %sp. Cette instruc tion, exécutée par l'appelé, permet d'effectuer les tâches 3, 4, 6.

- ret_and_restore

$$\text{c'est la séquence} \begin{cases} ret \\ restore \end{cases}$$

ret : provoque un branchement à l'adresse %i7+2. Or dans %i7, on a l'adresse du call. Donc on se branche à l'instruction qui suit l'instruction dans le delay slot du call. Assure la tâche 11.
La notion de delay slot est liée au pipeline d'instructions.
restore : fait tourner la fenêtre de registres de 1 en sens inverse du save. De ce fait, %fp et %sp sont restaurés à leur ancienne valeur dans p. C'est la tâche 10.

On donne la définition complète de la traduction d'une déclaration de procédure avec les hypothèses suivantes :

- l'appelant range toujours la valeur du prédecesseur statique de l'appelé dans %g1
- l'appelé range toujours la valeur de son prédecesseur statique à l'adresse %fp +1
- on commence à ranger le premier paramètre à l'adresse %fp -17, le deuxième à l'adresse %fp -18, etc... (pour l'instant, on oublie la possibilité de passer les 6 premiers paramètres par les registres).
- TAILLEPILE est toujours égale au moins à 23 + place des variables locales (23 = 6 mots d'adresses 0-23) Elle aura une valeur plus grande si, dans le corps de la procédure, il y a des appels de procédure nécessitant plus de 6 mots pour passer les paramètres.

code(procedure p(specs); vdecls; pdecls; corps, ρ, pi) :
  (A, ρ') := elab_spec(specs, ρ, -17, pi+1)
  (LP, x) := ρ(p)
  (B, ρ") := elab_vdecl (vdecls, ρ', 2, pi+1)
  code := Code_procs;
    LP : save %sp, TAILLEPILE, %sp
    st %g1, [%fp +1]
    code associé aux initialisations éventuelles des variables locales
    code(corps, ρ"', pi+1)
    ret
    restore

3.9.5 Traduction de la transmission des paramètres et de l'appel de procédure

Si x est un paramètre formel passé par référence, alors transmettre le paramètre effectif consiste à calculer son adresse et à la ranger à l'adresse relative affectée à x
Si x est passé par valeur, le paramètre effectif doit être une expression du même type. A l'appel, l'expression est évaluée et sa valeur est rangée dans la zone des paramètres selon l'adresse relative assignée à x.

transmettre_params(param; list_param, A, nom_appele, ρ, pi) :
si type_param(param, numero_param,nom_appele) = param_par_ref
alors
code$_G$(param, ρ, pi, r_x)
st r_x, [%sp - 17 -A]
t =1
sinon si c'est une variable structurée
code$_G$(param, ρ, pi, r_y)
t := nbcell(param)
copier t cellules à partir de l'adresse dans r_y dans la zone débutant à l'adresse A
sinon c'est une exp. de type simple
code$_D$(param, ρ, pi, r_y)
st r_y, [%sp -17-A]
t=1
transmettre_params(list_param, A+t, nom_appele, ρ, pi)

Remarque : le cas des tableaux dynamiques est laissé à titre d'exercice.

Exemple : Considérons le programme suivant :
procedure $q()$
...
$p(x_1,\ x_2,\ x_3)$
...
fin de q

Soit $\rho(p$ = (L, pi$_p$). Soit pi$_q$ la profondeur d'imbrication courante de q et soit la traduction de l'appel $p(x_1,\ x_2,\ x_3)$ dans l'environnement ρ avec pi$_q$

- si pi$_q$ < pi$_p$ alors p est plus profondément imbriquée que q. Ça signifie que p est déclarée dans q (sinon elle ne serait pas visible de p). Dans ce cas, le prédécesseur statique de p est égal à son prédécesseur dynamique.
- si pi$_q$ $\geq$ pi$_p$, alors p et q ont les mêmes procédures englobantes aux profondeurs d'imbrication 1, 2, ..., pi$_p$-1. Il suffit de remonter pi$_q$ - pi$_p$ liens statiques pour avoir l'adresse du prédécesseur statique de p.
  On l'obtient par lireadr(pi$_q$ - pi$_p$, 1) %fp, %g1

Le schéma de traduction est donc le suivant :
lireadr(pi$_q$ - pi$_p$, 1), %fp, %g1
transmettre_params($x_1,\ x_2,\ x_3$, 0, p, ρ, pi$_q$)
call L, 3

où L est l'adresse du début de code de p ($\rho(p)$ = (L, pi$_p$)

3.9.6 Accès aux variables et aux paramètres formels

- si x est une variable (locale ou non) ou un paramètre formel passé par valeur, supposons $\rho(x)$ = (b, pi$_x$)
  code$_G$(x r, ρ, pi, r_y) =
  lireadr(b, pi$_x$ - pi, %fp, %r_y)
  code$_M$(r, ρ, pi, r_y, x)

  Remarque : si var. locale, lireadr(b, pi$_x$-pi, %fp, %r_y) s'écrit aussi %r_y := %fp + b
- si x est un paramètre formel passé par référence,
  code$_G$(x r, ρ, pi, r_y) =
  lireval(b, pi$_x$ - pi, %fp, %r_y)
  code$_M$(r, ρ, pi, r_y, x)

Les fonctions

Pour simplifier, on suppose qu'une fonction ne peut retourner qu'une valeur scalaire comme en Pascal et plus particulièrement une valeur entière.

Dans le premier modèle, pour retourner une valeur, on suppose que
- la valeur de retour est rangée au sommet du bloc d'activation de l'appelé
- l'instruction de retour de l'appelant ne dépile pas tout le bloc d'activation de l'appelé ; elle laisse le premier élément du bloc, i.e. celui où se trouve la valeur retournée.

Dans le modèle du SPARC, on utilise la fenêtre de registres. L'appelé range toujours la valeur retournée dans son registre %i0. Par conséquent, l'appelant la récupère toujours dans son %o0.

3.10 Le programme principal

code(program vdecls; pdecls; body, ∅, 0) :
(B, ρ) = elab_vdecl(vdecls, ∅, 2, 1)
(ρ', Code_procs) = elab_pdecl(pdecls, ρ', 1)
code := Code_procs ;
LP : save %sp, TAILLEPILE, %sp
st %g1, [%fp +1]
code associé aux initialisations éventuelles des variables locales
code(body, ρ', 1)
ret
restore

3.11 Un exemple complet de traduction :

```
program prin;
  var x, y, z:integer;
  procedure p1(var a:integer; b:integer);
      var x,i:integer;
      t1:array[1..5] of integer;
      t2: array[1..7] of integer;
      procedure p11(c:integer; d:array[1..5] of integer;
                         var e:array[1..7] of integer);
          var y,t:integer;
          begin
             y := x + d[z];
             e[y] := c
          end;
      begin
        for i:=1 to 5 do t1[i] := x;
        t2[6] := z + a;
        p11(b, t1, t2)
      end;
  begin
    x := 4;
    z := 7;
    p1(x, y+2)
  end.
```

Hypothèse : un entier occupe 4 octets (une adresse aussi !)

p11: save %sp, 31, %sp
st %g1, [%fp+1] sauvegarde le PPS

{y:=x + d[z]}
r_{13} = %fp + 2 $\{\rho(y) = (2,\ 3)\}$
lireval(2, 1), %fp, r_8 $\{\rho(x) = (2,\ 2)\}$
r_9 := %fp - 21 $\{\rho(d) = (-21,\ 3)\}$
lireval(10, 2), %fp, r_{10} $\{\rho(z) = (10,\ 1)\}$
$r_{10} := r_{10} - 1$
$r_{10} := r_{10} * 4$
$r_{11} := r_9\ -\ r_{10}$ $r_{11} \leftarrow \rho$(d[z])
Attention! les adresses des paramètres vont dans le sens négatif.
ld [r_{11}], r_{12} valeur de d[z]
$r_0 := r_8\ +\ r_{12}$
st r_0, [r_{13}]

{e[y] := c}
ld [%fp - 41], r_{14} à l'adresse %fp - 41, on a $\rho(e)$
r_{15} := %fp + 2 $\rho(y) = (2,\ 3)$
ld [r_{15}], r_{15}
$r_{15} := r_{15} - 1$
$r_{15} := r_{15} * 4$
$r_{16} := r_{14}\ +\ r_{15}$
r_{17} := %fp - 17
ld [r_{17}], r_{18}
st r_{18}, [r_{16}]

ret_and_restore

Remarque :

ld [%fp - 41], r_{14} $\Longleftrightarrow \begin{cases} r_x := \%fp - 41 \\ ld\ [r_x],\ r_{14} \end{cases}$

p1: save %sp, 79, %sp   23 + (2 +5 + 7)*4 = 79
st %g1, [%fp+1]   sauvegarde le PPS

{for i :=1 to 5 do t1[i] := x}
r_{17} := %fp + 6 $\rho(i) = (6,\ 2)$
$r_{18} := 1$
st r_{18}, [r_{17}]
$r_{16} := 5$
L1: cmp r_{18}, r_{16}
bgt L2
{t1[i] := x}
r_{19} := %fp + 10 $\rho(t1) = (10,\ 2)$
$r'_{18} := r_{18} - 1$
$r'_{18} := r'_{18} * 4$
$r_{20} := r_{19} + r'_{18}$ $r_{20} \leftarrow \rho(t1[i])$
r_{21} := %fp + 2 $\rho(x) = (2,\ 2)$
ld [r_{21}], r_{21}
st r_{21}, [r_{20}]
{i := i + 1}
$r_{18} := r_{18} + 1$ pour la boucle
st r_{18}, [r_{17}]   on range la valeur à $\rho(i)$
ba L1

{t2[6] := z + a}
L2: r_{19} := %fp + 30 $\rho(t2) = (30,\ 2)$
$r_{50} := 5$ 5 = 6-1
$r_{50} := r_{50} * 4$
$r_{20} := r_{19} + r_{50}$
lireval(10, 1), %fp, r_{21} $\rho(z) = (10,\ 1)$
ld [%fp - 17], r_{22} à l'adresse %fp-17, on a l'adresse de a
ld [r_{22}], r_{23} $r_{23} \leftarrow$ valeur de a
$r_{24} := r_{21} + r_{23}$
st r_{24}, [r_{20}]

{p11(b, t1, t2)}   on transmet les paramètres
{on range la valeur de b à l'adresse %sp - 17
ld [%fp - 21], r_{25}
st r_{25}, [%sp - 17]
{on range les 5 éléments de t1 aux adresses %sp-21, ... %sp-37}
ld [%fp+10], r_{26} $\rho(t1) = (4,\ 2)$
st r_{26}, [%sp-21]
ld [%fp+14], r_{26}
st r_{26}, [%sp-25]
ld [%fp+18], r_{26}
st r_{26}, [%sp-29]
ld [%fp+22], r_{26}
st r_{26}, [%sp-33]
ld [%fp+26], r_{26}
st r_{26}, [%sp-37]
{on range l'adresse de t2 à %sp-41}
r_{27} := %fp+30 $\rho(t2) = (9,\ 2)$
st r_{27}, [%sp - 41]
call p11, 3
ret_and_restore

main: save %sp, 35, %sp
{x := 4}
r_2 := %fp + 2 $\rho(x) = (2,\ 1)$
$r_1 := 4$
st r_1, [r_2]

{z := 7}
r_3 := %fp + 10 $\rho(z) = (10,\ 1)$
$r_4 := 7$
st r_4, [r_3]

{p1(x, y+2)}
r_5 := %fp + 2 $\rho(x) = (2,\ 1)$
st r_5, [%sp - 17]   on range l'adresse de x à l'adresse indiquée par %sp - 17
ld [%fp + 6], r_6 on charge la valeur de y dans r_6
$r_6 := r_6 + 2$
st r_6, [%sp - 21]
call p1, 2
ret_and_restore

3.12 Conclusion

On a appris à traduire *à la main*, pour un langage à structure de blocs de la famille de Pascal, les différentes structures de données (leur accès mémoire en particulier) et les instructions.

Evidemment, pour concevoir un compilateur, ces traductions à la main ne sont pas envisageables. On doit pour toutes les constructions du langage, écrire des schémas de traduction automatique.

Exemple : prenons un exemple simple. On considère la portion de grammaire suivante qui reconnaît les expressions booléennes.
On va écrire un schéma de traduction pour cette portion de grammaire.

Pour cela, on définit un attribut synthétisé *code* associé à E et ayant pour paramètres la fonction d'adressage ρ et un nom de registre.

On utilise la fonction *DonnerRegistre* qui fournit un **nouveau** nom de registre et la fonction *NouvEtiq* qui génère un **nouveau** nom d'étiquette symbolique.

Remarques :

- *DonnerRegistre* est à écrire avec soin si on dispose d'un nombre limité de registres.
- *NouvEtiq* peut consister de façon simple à incrémenter un compteur d'étiquette.

On définit aussi l'attribut *branch* associé à *oprel* de type chaîne de caractères, et qui récupère le branchement adéquat.

$E \quad \rightarrow \quad E \text{ ou } E \mid E \text{ et } E \mid \text{non } E \mid (E)$

$\quad\quad \mid \quad \text{id } \mathit{oprel} \text{ id} \mid \text{vrai} \mid \text{faux}$

$\mathit{oprel} \quad \rightarrow \quad < \mid \leq \mid > \mid \geq \mid = \mid \neq$

$E \rightarrow \quad E_1 \text{ ou } E_2$

$\{r := \mathit{DonnerRegistre}\,;$

$\mathit{code}(E,\rho,r) := \mathit{code}(E_1,\rho,r_1) || \mathit{code}(E_2,\rho,r_2) ||' r := r_1 \text{ or } r_2\ '\}$

$E \rightarrow \quad E_1 \text{ et } E_2$

$\{r := \mathit{DonnerRegistre}\,;$

$\mathit{code}(E,\rho,r) := \mathit{code}(E_1,\rho,r_1) || \mathit{code}(E_2,\rho,r_2) ||' r := r_1 \text{ and } r_2\ '\}$

$E \rightarrow \quad \text{non } E_1$

$\{r := \mathit{DonnerRegistre}\,;$

$\mathit{code}(E,\ \rho,r) := \mathit{code}(E_1,\rho,r_1) ||' r := \text{not } r_1\ '\}$

$E \rightarrow \quad (E_1)$

$\{\mathit{code}(E,\ \rho,r) := \mathit{code}(E_1,\rho,r)\}$

$E \rightarrow \quad \text{id}_1 \ \mathit{oprel} \ \text{id}_2$

$\{\text{L}_1 := \mathit{NouvEtiq}\,; \text{suite} := \mathit{NouvEtiq}\,;$

$r_1 := \mathit{DonnerRegistre}\,;\ r_2 := \mathit{DonnerRegistre}\,;$

$r_3 := \mathit{DonnerRegistre}\,;$

$\mathit{code}(E,\rho,r_1) :=\ '\ r_2 := \rho(\text{id}_1)\ '\ ||\ '\ \text{ld}[r_2],\ r_2\ '||$

$'\ r_3 := \rho(\text{id}_2)\ '\ ||\ '\ \text{ld}[r_3],\ r_3\ '\ ||$

$'\ \text{cmp}\ r_2,\ r_3\ '\ ||\ \mathit{oprel.branch}\ ||\ '\ \text{L}_1\,;$

$r_1 := 1\,; \text{ba suite}\,; \text{L}_1\ :\ r_1 := 0\,; \text{suite}:\ '\}$

$E \rightarrow \quad \text{vrai}$

$\{r := \mathit{DonnerRegistre}\,;$

$\mathit{code}(E,\rho,r) :=\ '\ r := 1\ '\}$

$E \rightarrow \quad \text{faux}$

$\{r := \mathit{DonnerRegistre}\,;$

$\mathit{code}(E,\rho,r) :=\ '\ r := 0\ '\}$

$\mathit{oprel} \rightarrow \quad <\ \{\mathit{oprel.branch} :=\ '\ \text{bge}\ '\}$

$\mathit{oprel} \rightarrow \quad \leq\ \{\mathit{oprel.branch} :=\ '\ \text{bgt}\ '\}$

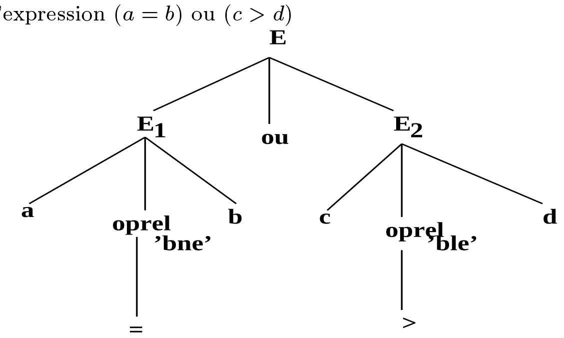
$\mathit{oprel} \rightarrow \quad >\ \{\mathit{oprel.branch} :=\ '\ \text{ble}\ '\}$

$\mathit{oprel} \rightarrow \quad \geq\ \{\mathit{oprel.branch} :=\ '\ \text{blt}\ '\}$

$\mathit{oprel} \rightarrow \quad =\ \{\mathit{oprel.branch} :=\ '\ \text{bne}\ '\}$

$\mathit{oprel} \rightarrow \quad \neq\ \{\mathit{oprel.branch} :=\ '\ \text{beq}\ '\}$

Soit l'expression $(a = b)$ ou $(c > d)$



$\mathit{code}(E,\rho,r) := \mathit{code}(E_1,\rho,r_1)\,; \rightsquigarrow r_2 := \rho(a)\,;\ \text{ld}[r_2],\ r_2\,;$

$r_3 := \rho(b)\,; \text{ld}[r_3], r_3\,;$

$\text{cmp}\ r_2, r_3\,; \text{bne L}_1$

$r_1 := 1\,;\ \text{ba suite}\,; \text{L}_1\ :\ r_1 := 0\,;$

$\text{suite}:$

$\mathit{code}(E_2,\rho,r_4)\,; \rightsquigarrow r_5 := \rho(c)\,; \text{ld}[r_5], r_5$

$r_6 := \rho(d)\,;\ \text{ld}[r_6], r_6\,; \text{cmp}\ r_5, r_6$

$\text{ble L}_2\ ;\ r_4 := 1\,; \text{ba suite'}\ ;$

$\text{L}_2\ :\ r_4 := 0\,; \text{suite'}:$

$r := r_1 \text{ or } r_4$

4. Production de code intermédiaire

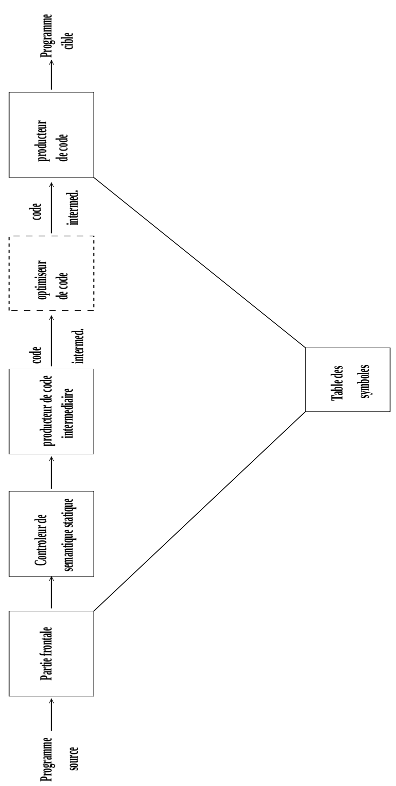
1. Les différentes structures de langages intermédiaires
   - Arbres abstraits décorés
   - DAG
   - Code à trois adresses
2. Représentation des programmes
   - Graphe de flot de contrôle
   - Graphe des blocs de base
   - Graphe des appels de procédure

Dans le modèle de compilation par *analyse-synthèse*, la partie frontale du compilateur traduit le programme source en une représentation intermédiaire (RI).
A partir de la RI, on produit du code objet pour la machine réelle (partie finale).

L'utilisation d'une forme intermédiaire indépendante de la machine cible présente les avantages suivants :

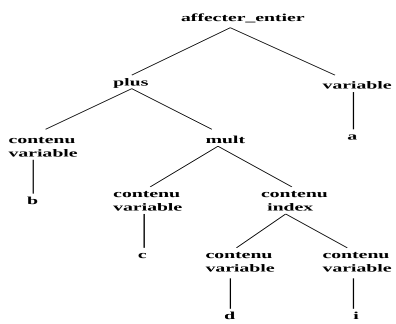
- le *reciblage* est facilité : on peut créer un compilateur pour une autre machine en modifiant seulement la partie finale du compilateur.
- un optimiseur de code indépendant de la machine peut être appliqué à la RI.

Situation du producteur de code intermédiaire :

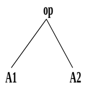


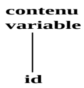
4.1 Les différentes structures de langages intermédiaires :
On en a vu une, le code SPARC virtuel. Idée : pas de noms pour les calculs intermédiaires, tout se fait par registres. A chaque structure correspond un ensemble de schémas de traduction différents.

- **Les arbres abstraits décorés.**
  Exemple : texte source a := b + c * d[i]



Exemple : un schéma de traduction pour la production d'arbres abstraits décorés représentant des instructions d'affectations.

Soient CreerNœud(op, A_1, A_2) = 

et CreerFeuille(contenu variable, id.entrée) = 

pnœud : attribut synthétisé, récupère la représentation intermédiaire sous forme d'arbre abstrait décoré.

id.entrée : entrée dans la TDS associée à id.

I → id := E
{I.pnœud := CreerNœud('Affecter_entier', E.pnœud,
CreerFeuille(variable, id.entrée))}

E → E_1 + T
{E.pnœud := CreerNœud('plus', E_1.pnœud, T.pnœud)}

E → T
{E.pnœud := T.pnœud}

T → T_1 * F
{T.pnœud := CreerNœud('mult', T_1.pnœud, F.pnœud)}

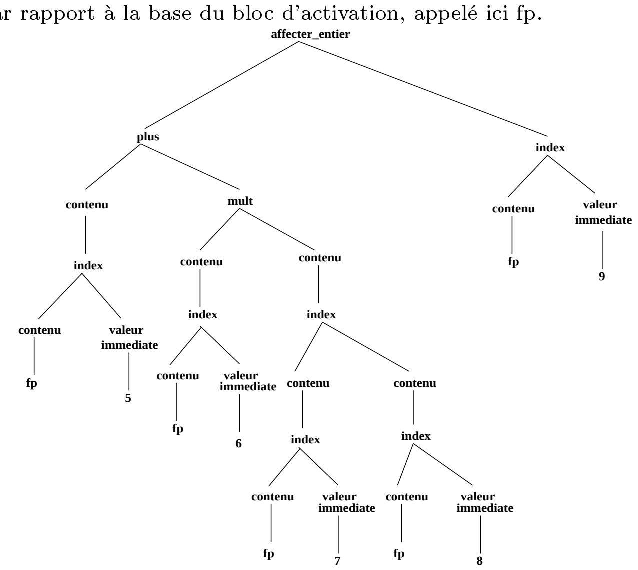
T → F
{T.pnœud := F.pnœud}

F → id
{F.pnœud := CreerFeuille(contenu variable, id.entrée)}

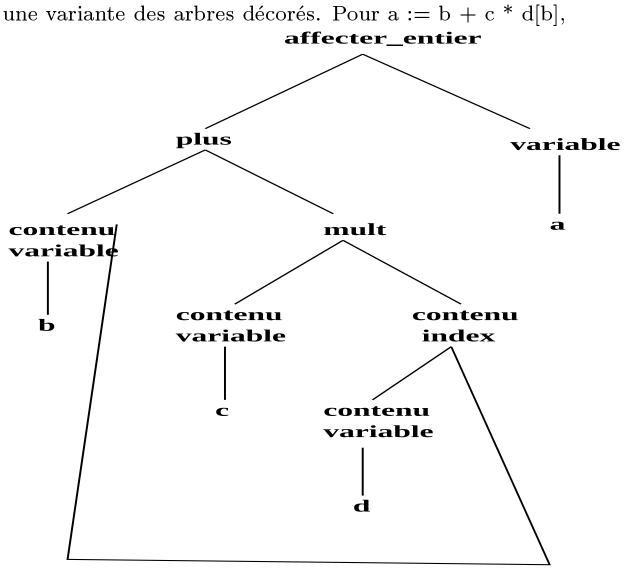
F → id [E]
{F.pnœud := CreerNœud('Contenu index',
CreerFeuille(contenu variable, id.entrée), E.pnœud)}

F → (E)
{F.pnœud := E.pnœud}

Cet arbre abstrait peut même être un arbre représentant des opérations plus proches de la machine ; en particulier, on peut avoir le chemin d'accès aux opérandes complètement expansé.
si $\rho(b) = 5,\ \rho(c) = 6, \ldots,\ \rho(a) = 9$,
Les variables disparaissent, et il ne reste plus que le chemin d'accès par rapport à la base du bloc d'activation, appelé ici fp.



- **Les DAG.**
c'est une variante des arbres décorés. Pour a := b + c * d[b],



Remarque : La variable b est répétée 2 fois.

- **Code à 3 adresses.**
  C'est une séquence d'instructions de la forme x := y op z
  où x, y, z sont des
  - des noms,
  - des constantes,
  - des variables temporaires produites par le compilateur.

  x est le résultat, op un opérateur.
  Exemple : a := b + c * d[i]. On traduit par :

```
t1 := c * d[i]
a := b + t1
```

Remarques :

- Il est plus facile d'étudier l'optimisation de code avec la représentation graphique du code à 3 adresses.
- l'utilisation de noms calculés par le programme pour les valeurs intermédiaires facilite les réarrangements du code à 3 adresses (toujours à des fins d'optimisation).

De façon plus précise, les instructions de code à 3 adresses sont les suivantes :

- x := y op z
  où op est un opérateur binaire arithmétique ou logique.
- x := op y
  où op est un opérateur unaire
  (le moins unaire, la négation logique, les opérateurs de décalage, les opérateurs de conversion comme la conversion d'un nombre en virgule fixe en un nombre en virgule flottante).
- x := y
  instruction de copie, la valeur de y étant affectée à x.
- allerà L
  branchement inconditionnel.
- si x oprel y allerà L
  oprel : $<$, $=$, $\geq, \ldots$
  Dans le cas où la condition n'est pas satisfaite, a pour effet de continuer en séquence.

- appels de procédure p(x_1, …, x_n)
  param x_1
  ...
  param x_n
  appeler p, n
  [return y]
- variables indicées
  - x := y[i]
    affecte à x la valeur de l'emplacement situé à i unités au-delà de la cellule y.
  - x[i] := y
    range dans l'emplacement situé à i unités de l'emplacement de x la valeur de y.
- instructions d'affectations d'adresses et de pointeurs
  - x := &y
    affecte à x l'adresse de y,
    y est un nom de temporaire qui dénote une expression ayant une valeur_g, x est un nom de pointeur ou un temporaire.
  - x := *y
    donne à la valeur_d de x la valeur du contenu de l'emplacement mémoire de y.
    y est un pointeur ou un temporaire.
  - *x := y
    affecte à la valeur_d de l'objet pointé par x la valeur_d de y.

Dans tous les cas, le choix des opérateurs de la RI est très important. Il doit être suffisamment riche pour permettre d'implanter tous les concepts du langage source.

Si petit nombre d'opérateurs, alors plus facile à implanter sur une machine cible,
Mais nécessite une production de longues séquences de traduction.
Dans ce cas, optimisation et génération de code très lourdes.

Exemple :
Soit A : array[1..10,1..20] of 4-entiers. Traduire l'affectation

$$x := A[y, z]$$

en code à 3 adresses.
Soit c la constante $\rho(A[1,1])$ - 84,
$\rho(A[y, z]) = \rho(A[1, 1]) - 84 + (y*d_2 + z) * 4 = c + (y*d_2+z)*4$
On aura le code suivant :

```
t1 := y * 20
t2 := t1 + z
t3 := t2 * 4
t4 := c
t5 := t4[t3]
x :=  t5
```


Remarque : une autre particularité de certains compilateurs : le code court-circuit (comme en Turbo-Pascal).
Le code court-circuit consiste à ne pas évaluer toute l'expression booléenne si on peut en déterminer la valeur avant toute l'évaluation.
Exemple : si a < b ou c < d et e < f

- si a < b est vraie, alors il n'est pas nécessaire d'évaluer le reste de l'expression pour savoir que l'expression est vraie.

- si a < b est fausse, et si c < d est fausse, il n'est pas utile d'évaluer e < f pour savoir que l'expression est fausse.

Traduction	complète	code	court-circuit
	si a < b allerà L_1		si a < b allerà L_7
	t1 := 0		si c $\geq$ d allerà L_8
	allerà L_3		si e < f allerà L_7
L_1:	t1 := 1		allerà L_8
L_3:	si c < d allerà L_2	L_7:	t9 := 1
	t2 := 0		allerà L_9
	allerà L_4	L_8:	t5 := 0
L_2:	t2 := 1	L_9:	
L_4:	si e < f allerà L_5		
	t3 := 0		
	allerà L_6		
L_5:	t3 := 1		
L_6:	t4 := t2 et t1		
	t5 := t1 ou t4		

Deux exemples de schémas de traduction pour produire du code intermédiaire

Exemple1 : code à 3 adresses pour les affectations.
code : attribut synthétisé, de type chaîne de caractères, destiné à contenir le code à 3 adresses produit par le schéma de traduction.
E.entrée prend pour valeur le nom du temporaire qui contient l'expression E. entrée est un attribut synthétisé.

Productions	Règles sémantiques
I → id := E	I.code := E.code || id.entrée || ':=' || E.entrée
E → E_1 + E_2	E.entrée := NouvTemp
	E.code := E_1.code || E_2.code ||
	E.entrée || ':=' || E_1.entrée || '+' || E_2.entrée
E → E_1 * E_2	E.entrée := NouvTemp
	E.code := E_1.code || E_2.code ||
	E.entrée || ':=' || E_1.entrée || '*' || E_2.entrée
E → - E_1	E.entrée := NouvTemp
	E.code := E_1.code || E.entrée || ':= -' || E_1.entrée
E → (E_1)	E.entrée := E_1.entrée
	E.code := E_1.code
E → id	E.entrée := id.entrée
	E.code := "

Exemple2 : code à 3 adresses pour l'instruction tant-que.

Production	Règle sémantique
I $\rightarrow$ tant que E faire I_1	I.début := NouvEtiq
	I.après := NouvEtiq
	I.code := I.début || ':' ||
	E.code ||
	'si' || E.entrée || '= 0 allerà' I.après ||
	I_1.code ||
	'allerà' I.début ||
	I.après || ':'

I.début et I.après sont deux nouveaux attributs pour marquer respectivement la première instruction engendrée pour le code de E et l'instruction suivant le code de I.
I.après prend la valeur de l'étiquette de l'instruction qui suit le code associé à l'instruction tant-que.
Ces attributs contiennent des valeurs d'étiquettes créées par la fonction NouvEtiq, qui renvoie une nouvelle étiquette chaque fois qu'elle est appelée.
On suppose qu'une expression non nulle représente la valeur *vrai*.
Par conséquent, quand la valeur de E devient égale à 0, le flot de contrôle quitte l'instruction tant-que.

2. Représentation des programmes

Blocs de base et graphes de flot de contrôle :
Une représentation sous forme de graphe des instructions à 3 adresses appelée *graphe de flot de contrôle (gfc)* facilite la compréhension des algorithmes de production de code.

Les nœuds d'un gfc représentent des calculs et les arcs représentent le flot de contrôle.

Définition. Bloc de base :
c'est une séquence d'instructions consécutives dans laquelle le flot de contrôle est activé au début de celle-ci et inhibé à sa fin sans possibilité d'arrêt ou de branchement autre qu'à la fin de la séquence.

Définition :
Une instruction à 3 adresses de la forme $x := y + z$ définit x et utilise (ou référence) y et z.
Un nom dans un bloc de base est actif en un point donné si sa valeur est utilisée après ce point dans le programme, éventuellement dans un autre bloc de base.

Définition. Instruction de tête :

- toute instruction qui peut être atteinte par une instruction de branchement conditionnel ou inconditionnel est une instruction de tête ou
- toute instruction qui suit immédiatement un branchement conditionnel ou inconditionnel.

L'algorithme suivant partitionne une séquence d'instructions à 3 adresses en blocs de base :

Données : une séquence d'instructions ;
Résultat : une liste de blocs de base telle que chaque instruction à 3 adresses appartient exactement à un bloc.

Méthode :

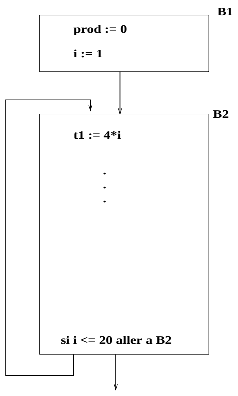
1. on détermine l'ensemble des instructions de tête de chaque bloc (la première instruction de la séquence est considérée comme une instruction de tête).
2. pour chaque instruction de tête, le bloc de base qui lui correspond débute par celle-ci et il est constitué de toutes les instructions suivantes qui ne sont pas une autre instruction de tête, ni la fin du programme.

Exemple : portion de programme qui calcule le produit scalaire des vecteurs a et b de longueur 20.

```
debut
  prod := 0;
  i := 1;
  tq i <= 20 faire
     prod := prod + a[i] * b[i];
     i := i+1
  ftq;
fin.
```

Code à 3 adresses correspondant :

1.	prod := 0	7.	$t_5 := t_2 * t_4$
2.	$i := 1$	8.	$t_6 := prod + t_5$
3.	$t_1 := 4 * i$	9.	$prod := t_6$
4.	$t_2 := a[t_1]$	10.	$t_7 := i + 1$
5.	$t_3 := 4 * i$	11.	$i := t_7$
6.	$t_4 := b[t_3]$	12.	si $i \leq 20$ aller à (3)



on a construit un graphe de flot de contrôle.

GFC. Définition :
C'est un graphe orienté dont les nœuds sont les blocs de base. On distingue un nœud particulier, le nœud initial : c'est le bloc dont l'instruction de tête est la première instruction.
Un arc orienté joint le bloc B1 au bloc B2 si B2 peut suivre immédiatement B1 dans une séquence d'exécution, c'est-à-dire :

1. si la dernière instruction de B1 est une instruction de branchement conditionnel ou inconditionnel à la première instruction de B2 ou
2. si B2 suit immédiatement B1 dans l'ordre du programme et B1 ne se termine pas par un branchement inconditionnel.

On dit que B1 est prédécesseur de B2 et que B2 est successeur de B1.

$\mathcal{P}(B)$ = ensemble des prédécesseurs de B,

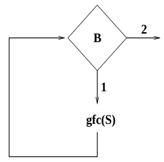
$\mathcal{S}(B)$ = ensemble des successeurs de B.

Les nœuds possédant plus d'un prédécesseur sont appelés des **jonctions**, et ceux qui possèdent plusieurs successeurs sont des **embranchements**.

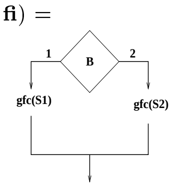
On donne ainsi une autre définition d'un bloc de base :
Définition. Bloc de base : c'est un chemin de longueur maximale, contenant au plus une jonction au début et un embranchement à la fin.

Gfc pour instructions composées

cfg(**while** B **do** S **od**) =



cfg(**if** B **then** S_1 **else** S_2 **fi**) =

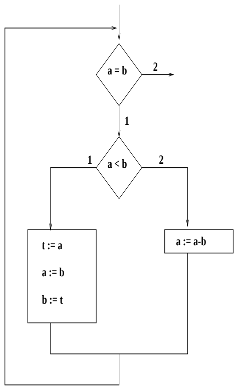


cfg(S_1 ; S_2) =

$$\begin{array}{c}\text{gfc(S1)}\\ \downarrow \\ \text{gfc(S2)}\end{array}$$

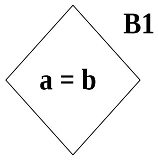
Exemple :

while a ≠ b **do**
  if a < b
  then t := a; a := b; b := t
  else a := a - b
  fi
od

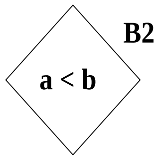


Le partitionnement en blocs de base :

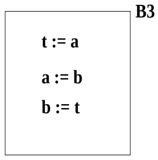
Le bloc B1 :



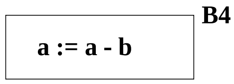
Le bloc B2 :



Le bloc B3 :



Le bloc B4 :



Graphe des appels : il représente la relation entre les procédures.

Définition le graphe des appels contient un nœud pour le programme principal (c'est le nœud d'entrée) et un nœud pour chaque procédure déclarée dans le programme. Les nœuds sont étiquetés par les noms de procédures.
Un arc conduit du nœud de p à celui de q si p contient un appel à q.

Le graphe des appels est utilisé pour les analyses inter-procédurales des programmes.

- il permet de voir facilement si une procédure est directement récursive ou mutuellement récursive avec d'autres procédures,
- utilisé pour détecter certains effets produits par un appel de procédure (ex. les variables susceptibles d'être modifiées par cet appel, ou les variables que la procédure pourrait utiliser).

5. Génération de code

1. La machine réelle
   (a) Hiérarchie de mémoire
   (b) les registres
   (c) Le jeu d'instructions
   (d) Les modes d'adressage
   (e) Le mécanisme d'évaluation des expressions
2. Classification des architectures
   (a) CISC
   (b) RISC
   Parallélisme intra-processeur
   i. VLIW
   ii. Pipeline d'instructions.

3. Un générateur de code simple
4. Les sous-tâches de la génération de code
   (a) la sélection d'instructions
   (b) L'allocation de registres
      i. Allocation globale
      ii. Assignation des registres pour les boucles externes
      iii. Allocation de registres par coloriage de graphes

5.1 La machine réelle. Quelques caractéristiques communes aux ordinateurs réels les plus courants.

- **Hiérarchie de mémoire**, déterminée par la rapidité des accès et la durée de vie des objets accédés.
  exemple : les registres du processeur, un cache, une mémoire principale et une mémoire auxiliaire.

  Un accès à la mémoire s'effectue *en général* à travers le bus système entre le processeur et la mémoire à l'aide d'un circuit qui sélectionne les cellules mémoire adressées. Tandis que les registres se trouvent généralement sur la puce du processeur.
  ⇒ différence considérable dans les temps d'accès.
  ce qui peut être intéressant : maintenir autant que possible les objets dans des registres.

- **Les registres du processeur**. Les classes usuelles :
  – les **registres universels**, pour des opérations arithmétiques (souvent limitées aux opérations sur les entiers), ou logiques, et pour des calculs d'adresses.
  – les **registres flottants**, pour les opérations en virgule flottante.
  – plutôt que des registres universels, quelquefois, ce sont des **registres de données** et des **registres d'adresses** séparés le MC68000 : 8 registres de données, et 8 regitres d'adresses).
  – **les registres de base**, pour adresser des segments de la mémoire, comme les blocs d'activation, et des registres d'index dont le contenu pourra être combiné avec d'autres adresses pour adresser des composants de tableaux.

  Et d'autres registres du processeur employés implicitement dans les opérations.

  – le **compteur d'instructions (PC)**. Implicitement utilisé au chargement d'une instruction, et modifié lors des ruptures de cycle (instructions de branchement, appels de procédure).
  – le **code condition**, pour ranger le résultat des comparaisons (souvent avec 0).
  Le CC contient en général des bits réservés à la retenue dans les additions (C *carry*), au débordement (V), au test de nullité (Z) et de signe (positif ou négatif) (N).

- **Jeu d'instructions :** celui d'un calculateur universel réel doit supporter plusieurs classes de langages et d'applications.
  Le nombre d'instructions peut varier d'une petite trentaine à plus d'un millier.
  Instructions simples ou au contraire très complexes. Plusieurs catégories :
  – les instructions de calcul, (arithmétiques et booléennes) différenciées selon la longueur des opérandes.
  – les instructions de transfert, chargent les objets depuis la mémoire dans les registres, écrivent des registres en mémoire, assurent le transfert des objets entre 2 registres ou entre 2 cellules mémoire.
  – les branchements et les instructions de gestion des appels de procédures,
  – les instructions de communication avec le système d'exploitation et les périphériques.

  La taille du jeu d'instructions est à peu près égale au produit du nombre d'opérations (en comptant les variantes par longueur et type d'opérandes différents) par le nombre d'adressages autorisés.

- **Les modes d'adressage**. Sur certains processeurs, l'adressage des opérandes s'effectue à l'aide de modes d'adressage complexes ; sur d'autres processeurs, les opérandes devront toujours se trouver dans des registres.
  Le tableau suivant montre quelques exemples de modes d'adressage :

Mode	Format	Adresse	Coût additionnel
absolu	M	M	1
registre	R	R	0
indexé	c(R)	c + contenu(R)	1
registre indirect	*R	contenu(R)	0
indirect indexé	*c(R)	contenu(c+contenu(R))	1
littéral	#c	c	1

- **Evaluation des expressions.** La plupart des machines réelles actuelles sont des machines à registres et non des machines à pile : les opérandes et les résultats des opérations se trouvent soit dans des registres soit en mémoire principale. L'une des tâches de la génération de code est de déterminer où doivent se trouver les résultats intermédiaires. Si on peut les stocker dans des registres, on accélère les accès :
  - utiliser le plus possible les registres pour le stockage des valeurs des variables et des résultats intermédiaires, pour le calcul d'adresses, pour la transmission des paramètres et pour le retour des valeurs des fonctions ;
  - choisir des séquences d'instructions machine pour les constructions du langage, et privilégier les instructions les plus efficaces correspondant à des cas particuliers ;
  - exploiter les possibilités de parallélisme offertes par le processeur cible, par un ordonnancement judicieux des instructions indépendantes.

  L'importance et la difficulté de ces tâches varient avec les caractéristiques du processeur cible.
- Une notation pour les instructions. Les langages assembleurs fournis par les fabricants ne sont pas des modèles de lisibilité ni d'uniformité.

5.2 Classification des architectures :

- **CISC (Complex Instruction Set Computer)**
  Elles offrent des instructions complexes pour traiter les accès aux structures de données et l'organisation modulaire du programme et elles sont caractérisées par les éléments suivants :
  - des modes d'adressage nombreux et complexes pour traiter efficacement les accès aux données structurées telles que les tableaux, les structures, les listes et les blocs d'activation.
  - une grande variété d'opérations pour des opérandes de tailles différentes et des combinaisons diverses des types des opérandes et des résultats.
  - des temps d'exécution variables pour les instructions.
  - peu de registres.

La réalisation de ces jeux d'instruction exige d'utiliser soit une grande quantité de matériel, soit de gros micro-programmes. Avec la densité croissante d'intégration, il faut prendre en compte les besoins en espace d'une logique d'instructions complexes, et il reste en général peu de place pour les registres du processeur.

Les représentants les plus en vue de la famille CISC sont les processeurs de la série VAX, les Intel 80x86, les Motorola 680x0 et les NSC 32xxx.

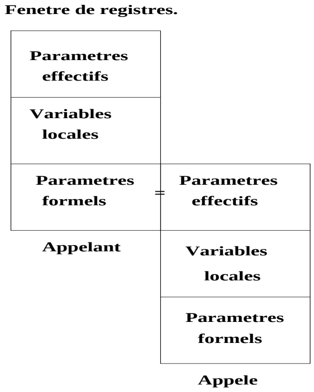
A cause du petit nombre de registres, la réalisation d'une bonne allocation de registres est une tâche très importante, mais difficile.

Comme un programme en langage source (ou en langage intermédiaire) admet plusieurs traductions correctes sur un processeur CISC (grande variété d'opérations), mais avec des performances qui peuvent être très différentes, une bonne sélection de code basée sur une estimation de coût est également extrêmement importante.

- **RISC (Reduced Instruction Set Computer)**
  Idée : accélérer l'exécution de chaque instruction individuellement grâce à une simplification de ces instructions.
  Les caractéristiques des architectures RISC sont les suivantes :
  - une instruction machine est exécutée à chaque cycle. Comme certaines instructions durent plus d'un cycle (ex. les accès à la mémoire), l'exécution utilise un traitement pipeliné.
  - les opérations de calcul n'accèdent qu'à des opérandes dans des registres, et déposent leurs résultats dans des registres (architectures *load/store*).
  - faible nombre de modes d'adressage.
  - le contrôle n'est pas micro-programmé, mais câblé.

  Donc, petit nombre d'instructions relativement simples,
  circuits moins encombrants ⇒ place récupérée sur la puce,
  cette place est réutilisée pour des registres, des unités fonctionnelles travaillant en parallèle, ou des circuits de communication.

Les processeurs RISC ont fréquemment plus d'une centaine de registres, qui sur certains processeurs ne sont accessibles qu'à travers une fenêtre d'accès. Une telle organisation permet d'implanter les appels de procédure et de fonctions :



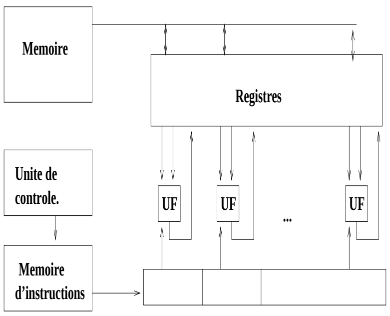
Intérêt : éviter la recopie du contenu de registres.

Le mode d'organisation des registres a une influence cruciale sur l'algorithme d'allocation de registres.

La sélection de code est rendue plus *ardue* par l'extrême simplicité du jeu d'instructions.

Parallélisme intra-processeur
Les processeurs RISC n'ont besoin que d'une surface assez réduite sur la puce pour implanter les circuits des instructions. La surface disponible est utilisée pour réaliser des unités fonctionnelles travaillant en parallèle ou des *pipelines* d'instructions ou de données (comme dans les caculateurs *vectoriels*).

- **VLIW (Very Long Instruction Words)** Soit un processeur équipé de plusieurs unités fonctionnelles travaillant en parallèle, relié à un grand nombre de registres par deux *portes d'entrée*, et une *porte de sortie* pour chaque unité. Le nombre de connexions aux registres permet le chargement simultané des opérandes par les unités fonctionnelles :



Le calculateur contient un flot d'instructions, un pointeur d'instructions et une unité de contrôle. Chaque mot d'instruction est suffisamment long pour contenir des sous-instructions destinées à toutes les unités fonctionnelles. Les temps d'instruction sont courts et prédictibles statiquement.

Travail du compilateur : rassembler le code généré pour un programme source dans une séquence de mots d'instructions longs ou très longs sans déformer la sémantique du programme.

Réordonner en mots d'instructions longs les séquences produites est difficile, particulièrement lorsqu'en réordonnant, on incorpore des séquences qui se trouvent au-delà d'instructions de branchement.

Nécessité d'effectuer au préalable une analyse de dépendances.

- **Pipeline d'instructions :**
  Dans un pipeline d'instructions peuvent se trouver à un instant donné plusieurs instructions à diverses phases de leur exécution. Un partage d'une instruction en 4 phases pourrait être le suivant :

  1. charger et décoder l'instruction
  2. charger les opérandes
  3. exécuter l'instruction
  4. écrire le résultat dans un registre.

  A chaque phase correspond un étage du pipeline responsable de son exécution, et l'exécution d'un étage prend un cycle. Cette division en étages permet de diminuer le temps de cycle. Dans une situation idéale :

		1	2	3	4	5	6	7
Etages	1	B_1	B_2	B_3	B_4			
du	2		B_1	B_2	B_3	B_4		
pipeline	3			B_1	B_2	B_3	B_4	
	4				B_1	B_2	B_3	B_4

Le pipeline n'est possible que si

1. les instructions successives de demandent pas simultanément les mêmes ressources matérielles (collision) et,
2. si aucune instruction n'a besoin du résultat d'une instruction précédente, et qui n'est pas encore disponible.

pipeline interlock : situation 2. L'alimentation du pipeline est retardée jusqu'à ce que la valeur requise par l'instruction devienne disponible.
Coûteux ⇒ plus d'actualité.
Le générateur de code réordonne les instructions de façon à intercaler entre deux instructions dépendantes un nombre suffisant d'instructions qui n'en dépendent pas, et éventuellement remplir les délais non utilisés par des instructions *NOP (No Operation)*.

Pour l'exécution d'un branchement, le pipeline reste vide pendant quelques cycles.

3. Un générateur de code simple

Produire du code cible pour une séquence d'instructions à 3 adresses.

Hypothèses :

1. A chaque opérateur d'une instruction de code intermédiaire correspond une opération de la machine cible,
2. on peut conserver les résultats de calculs dans des registres aussi longtemps que possible, c'est-à-dire on range ces résultats en mémoire uniquement
   - si on a besoin du registre qui les contient pour effectuer un autre calcul ou
   - juste avant un appel de procédure, une instruction de branchement ou une instruction étiquetée.
     ⇒ les contenus de tous les registres doivent être sauvegardés à la fin d'un bloc de base.

Conséquence : l'algorithme de production de code simple effectue toutes les sauvegardes nécessaires quand on change de bloc et quand on traite un appel de procédure.

1. La machine cible : machine adressable par octets avec des mots de 4 octets et n registres généraux R_0, $R_1, \ldots, R_n$.
Elle possède des instructions à 2 adresses de la forme

op source, destination

Exemple : MOV #1, R_0 charge la constante 1 dans le registre R_0

2. Descripteurs de registres et d'adresses : l'algorithme de production de code utilise des descripteurs pour garder trace des noms contenus dans les registres ou les adresses.

- Un descripteur de registres garde la trace du contenu courant des registres. On consulte le descripteur chaque fois qu'on a besoin d'un nouveau registre.
- Un descripteur d'adresses garde la trace du contenu d'un emplacement (ou des emplacements) où on peut trouver la valeur courante d'un nom à l'exécution. Cet emplacement peut être un registre, une adresse mémoire ou un ensemble de ces emplacements. Cette information peut être stockée dans la TDS et utilisée pour déterminer le mode d'accès à un nom.

Exemple : soit à traduire l'instruction

$$d := (a\text{-}b) + (a\text{-}c) + (a\text{-}c)$$

où d est active à la fin du bloc. La traduction en RI est :

t_1 := a-b
t_2 := a-c
t_3 := t_1 + t_2
d := t_3 + t_2

instruction	code produit	descripteur de registres	descripteur d'adresses
t_1 := a-b	MOV a, R_0 SUB b, R_0	R_0 contient t_1	t_1 est dans R_0
t_2 := a-c	MOV a, R_1 SUB c, R_1	R_0 contient t_1 R_1 contient t_2	t_1 est dans R_0 t_2 est dans R_1
t_3 := t_1 + t_2	ADD R_1, R_0	R_0 contient t_3 R_1 contient t_2	t_2 est dans R_1 t_3 est dans R_0
d := t_3 + t_2	ADD R_1, R_0 MOV R_0, d	R_0 contient d	d est dans R_0 et dans la mémoire

Remarques :

- le fait que a, b et c sont toujours présents dans la mémoire ne figure pas dans le descripteur d'adresses.
- R_1 devient vide car t_2 n'a pas d'utilisation ultérieure.

coût du code = 12

L'algorithme de production de code : pour chaque instruction à 3 adresses de la forme x := y op z,

1. invoquer la fonction DonnerRegistre pour déterminer l'emplacement E dans lequel le résultat du calcul y op z sera rangé ;
2. consulter le descripteur d'adresses pour déterminer l'emplacement courant y' de y (ou l'un de ses emplacements courants). Choisir de préférence un registre.
   Si y n'est pas déjà dans E, produire MOV y', E ;
3. produire l'instruction OP z', E où z' est un des emplacements courants de z. Préférer un registre si possible.
   Modifier le descripteur d'adresses pour indiquer que x est dans E.
   Si E est un registre, modifier son descripteur pour indiquer qu'il contient x ;
4. si les valeurs courantes de y et/ou z n'ont pas d'utilisation ultérieure, ne sont pas actives à la sortie du bloc, et sont dans des registres, modifier le descripteur de registres pour indiquer qu'après l'exécution de x := y op z, les registres ne contiennent plus y et/ou z.

Quand toutes les instructions à 3 adresses d'un bloc de base ont été traitées, on range par des instructions MOV, les noms qui ne sont pas dans des emplacements mémoire et qui sont actifs à la sortie du bloc. (on utilise l'information sur les variables actives).

5.4 Les sous-tâches de la génération de code

Objectif : le code produit doit être correct et de bonne qualité (le meilleur usage possible des ressources de la machine).
De plus, rapidité d'exécution.

Remarque : En théorie, le problème de la production de code optimal est indécidable. En pratique, on utilise des heuristiques qui produisent un bon code pas nécessairement optimal.

Donnée du générateur de code : une représentation intermédiaire du programme source, couplée avec la TDS.

Résultat : le programme cible sous l'une des 3 formes suivantes :

1. langage machine absolu,
2. langage machine translatable,
3. langage d'assemblage.

- Code machine absolu.
  Avantage : peut être chargé dans un emplacement donné de la mémoire et immédiatement exécuté.
- Code machine translatable.
  Avantage : permet la compilation séparée des sous-programmes. L'édition de liens et le chargement d'un ensemble de modules objets translatables permettent de préparer l'exécution.
  Coût supplémentaire : celui des phases d'édition de liens et de chargement.
  Souplesse d'utilisation.
- Code assembleur.
  Production de code plus simple. On peut produire des instructions symboliques et utiliser les facilités des macros de l'assembleur.
  Coût supplémentaire : celui de la phase d'assemblage après la production de code.

On choisit la 3ème option pour des raisons de lisibilité.

(a) La sélection d'instructions

d'elle dépend le coût d'exécution du code.
En outre, elle joue un rôle particulier dans le cas des machines CISC sur lesquelles on a le choix entre plusieurs possibilités pour produire une portion de code.

Exemple : soit à produire du code pour l'instruction

$$b := 2 + a[i]$$

où b et i sont des variables entières et a un tableau[0..] of integer.
On suppose que le registre R_0 est utilisé comme un registre d'adresse et qu'il contient l'adresse de début du bloc d'activation.
Soit $\rho(b) = -4$, $\rho(i) = -6$ et $\rho(a) = -8$.
La séquence d'instructions de code à 3 adresses est la suivante :

$t_1 := a[i]$
$t_2 := 2 + t_1$
$b := t_2$

On peut produire 3 portions différentes de code, avec des coûts différents :

1. MOV -8(R_0), R_1
   MOV *-6(R_0), R_2
   ADD R_2, R_1
   MOV #2, R_2 — coût = 10
   ADD *R_1, R_2
   MOV R_2, -4(R_0)
2. MOV -6(R_0), R_1
   MOV *R_1, R_1
   MOV -8(R_0), R_2
   ADD R_1, R_2
   MOV *R_2, R_2
   MOV #2, R_1 — coût = 13
   ADD R_2, R_1
   MOV -4(R_0), R_2
   MOV R_1, R_2
3. MOV #-6, R_1
   ADD R_0, R_1
   MOV *R_1, R_1
   MOV #-8, R_2
   ADD R_0, R_2
   ADD R_2, R_1 — coût = 16
   MOV *R_1, R_1
   MOV #2, R_2
   ADD R_1, R_2
   MOV #-4, R_1
   ADD R_0, R_1
   MOV R_2, R_1

(b) L'allocation de registres

Une utilisation efficace des registres est un critère important pour la production de code. En effet, les instructions dont les opérandes sont des registres sont plus courtes et plus rapides que celles dont les opérandes sont en mémoire.

2 aspects du problème :

1. quelle(s) valeur(s) du programme doivent résider dans des registres ? (allocation de registres),
2. dans quel registre chaque valeur doit résider ? (assignation de registres).

Une approche consiste à assigner des valeurs spécifiques du programme objet à certains registres. Ex : assigner les adresses de base à un groupe de registres, les calculs arithmétiques à un autre, ...
Avantage : simplifier la conception du compilateur,
Inconvénient : si elle est appliquée de façon systématique, les registres ne sont pas utilisés efficacement (des registres peuvent rester inutilisés même en cas de besoin).

Allocation globale des registres.

Stratégie : assigner un nombre déterminé de registres à la conservation des valeurs les plus utilisées dans chaque boucle interne.
Pourquoi : parce que les programmes passent la plus grande partie de leur temps à exécuter les boucles internes.
Pour le moment, on suppose connue la structure des boucles d'un gfc et qu'on sait quelles valeurs calculées dans un bloc sont utilisées à l'extérieur de ce bloc.
Remarque : Les valeurs sélectionnées peuvent appartenir à différentes boucles.
Les registres restants peuvent être utilisés pour contenir les valeurs locales à un bloc.
Pas nécessairement la meilleure allocation globale de registres. Mais méthode simple à implanter (Fortran H, compilateur optimisant de la série de machines IBM/360).

Allocation globale des registres.

Décomptes d'utilisation : méthode pour déterminer les économies réalisées en conservant une variable x dans un registre pendant la durée de la boucle C.

- économie d'une unité pour toute référence à x si x est dans un registre.
- économie d'une unité pour chaque utilisation de x dans la boucle C qui n'est pas précédée par une affectation de x dans le même bloc,
- économie de 2 unités si on peut éviter un rangement de x en mémoire à la fin d'un bloc, si x est actif à la sortie de ce bloc et si une valeur lui est affectée.
- coût d'une unité si x est actif à l'entrée de C et qu'on doit la charger dans un registre.

Hypothèse : C est exécutée un grand nombre de fois. L'économie, si on range x dans un registre peut être approchée par la formule

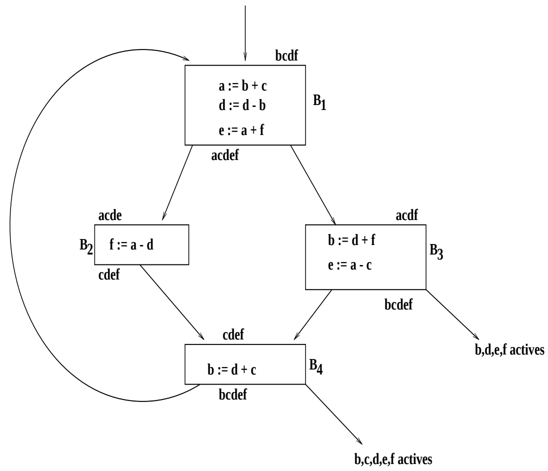
$$\Sigma_{\text{blocs B}}(Util(x,\ \mathrm{B}) + 2 * Actif(x,\ \mathrm{B}))$$

$Util(x,\ \mathrm{B})$ = nombre de fois où x est utilisée dans B avant toute définition de x,

$$Actif(x,\ \mathrm{B}) = \begin{cases} 1 \text{ si } x \text{ est actif à la sortie de B et si } x \text{ est défini dans B} \\ 0 \text{ sinon} \end{cases}$$

Allocation globale des registres.

Exemple :



$Util(a,\ \mathrm{B}_1) = 0, Util(a,\ \mathrm{B}_2) = Util(a,\ \mathrm{B}_3) = 1$. $Util(a,\ \mathrm{B}_4) = 0$. $\Sigma_B\ Actif(a,\ \mathrm{B}) = 2$. Donc, l'économie de coût si on range a dans un registre est de 4.

variable	a	b	c	d	e	f
économie	4	6	3	6	4	4

On peut choisir de ranger a (ou e ou f), b et d dans des registres.

Allocation globale des registres.

Exemple :(suite)

B_1	MOV b, R_0	MOV c, R_1	ADD R_1, R_0	MOV R_0, a
	MOV d, R_0	MOV b, R_1	SUB R_1, R_0	MOV R_0, d
	MOV a, R_0	MOV f, R_1	ADD R_1, R_0	MOV R_0, e
B_2	MOV a, R_0	MOV d, R_1	SUB R_1, R_0	MOV R_0, f
B_3	MOV d, R_0	MOV f, R_1	ADD R_1, R_0	MOV R_0, b
	MOV a, R_0	MOV c, R_1	SUB R_1, R_0	MOV R_0, e
B_4	MOV d, R_0	MOV c, R_1	ADD R_1, R_0	MOV R_0, b

coût total = 3*7 (B_1) + 7 (B_2) + 2*7 (B_3) + 7 (B_4) = 49.

Allocation globale des registres.

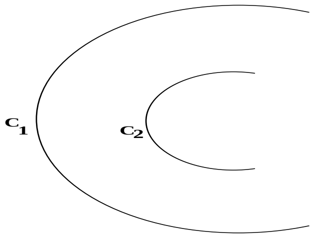
Exemple :(suite)

Si R_0 est dédié à b :

B_1	MOV c, R_1	MOV R_0, R_2	ADD R_1, R_2	MOV R_2, a
	MOV R_0, R_2	MOV d, R_1	SUB R_2, R_1	MOV R_1, d
	MOV a, R_0	MOV f, R_1	ADD R_1, R_0	MOV R_0, e
B_2	MOV a, R_0	MOV d, R_1	SUB R_1, R_0	MOV R_0, f
B_3	MOV d, R_0	MOV f, R_1	ADD R_1, R_0	
	MOV a, R_1	MOV c, R_2	SUB R_2, R_1	MOV R_1, e
B_4	MOV d, R_0	MOV c, R_1	ADD R_1, R_0	

coût total = 6+6+7 (B_1) + 7 (B_2) + 5+7 (B_3) + 5 (B_4) = 43.

Assignation de registres pour les boucles externes.



- les noms pour lesquels on a alloué des registres dans C_2 n'ont pas besoin de se voir allouer un registre dans C_1 - C_2 ;
- si on a alloué un registre pour x dans C_1, mais pas pour C_2, on doit sauvegarder x à l'entrée de C_2, et la recharger si on quitte C_2 et si on entre dans C_1 - C_2 ;
- si on alloue un registre à x dans C_2 mais pas dans C_1, on doit charger x à l'entrée de C_2 et la sauvegarder à sa sortie.

On peut trouver un critère de sélection des noms pour lesquels des registres doivent être alloués dans une boucle externe C, étant donné que les choix ont déjà été faits pour toutes les boucles imbriquées à l'intérieur de C.

Allocation de registres par coloriage de graphes.

Jusqu'à présent, ce qu'on a vu : répartition astucieuse des registres.

L'allocation de registres par coloriage de graphes : c'est une technique d'allocation de registres qui attribue les registres de la machine globalement pour tous les calculs à l'intérieur d'une procédure.

Donnée : programme en langage intermédiaire, où chaque opération et chaque variable ont été affectées d'un registre symbolique.

But : faire correspondre aux registres symboliques (dont le nombre n'est pas borné) des registres de la machine réelle, qui sont en nombre fini.

Définition. Durée de vie d'un registre.
Un registre symbolique r est vivant en un point donné p du programme s'il reçoit une valeur sur un chemin menant du point d'entrée de la procédure au point p, et s'il existe un chemin depuis p à une utilisation de r, sur lequel r n'est pas redéfini.
La durée de vie d'un registre symbolique r est l'ensemble des points du programme sur lequel r est vivant.

r est donc vivant en p si sa valeur doit être réutilisée plus tard.

Allocation de registres par coloriage de graphes.

Définition. Graphe de collision des registres
Deux durées de vie de registres symboliques sont en collision si l'un de ces registres est affecté au cours de la durée de vie de l'autre.
Le graphe de collision des registres est un graphe non orienté. Ses nœuds sont les durées de vie des registres symboliques et un arc joint deux durées de vie en collision.

Le problème d'allocation des regsitres devient : colorier le graphe de collisions avec k couleurs, où k est le nombre de registres réels.
Le coloriage ne doit pas donner la même couleur à deux nœuds connectés directement.

Si $k > 2$, décider si un graphe est k-coloriable est un problème NP-complet.

Si le graphe de collisions n'est pas k-coloriable, l'algorithme d'allocation utilise une heuristique pour décharger un registre en mémoire.

6. Optimisation de code

1. Introduction
   (a) Critères pour optimiser le code
   (b) place des tâches d'optimisation
2. Les principales sources d'optimisation
   (a) Sous-expressions communes
   (b) Propagation des copies
   (c) Elimination du code inutile
   (d) Optimisation de boucles
   (e) Techniques implantées pour effectuer les transformations.
3. Analyse globale de flot de données
   (a) Equations
   (b) Réordonnancement des instructions

6.1 Introduction

Il est rare que les compilateurs produisent du code cible d'aussi bonne qualité que celui écrit à la main.

Pour rendre le code produit plus rapide et/ou plus compact, on effectue des transformations sur le code appelées *optimisations*,

- optimisations indépendantes de la machine : améliorent le code cible sans tenir compte des propriétés de la machine cible ;
- optimisations dépendantes de la machine : telles que l'allocation de registres, l'utilisation de séquences d'instructions spécifiques.

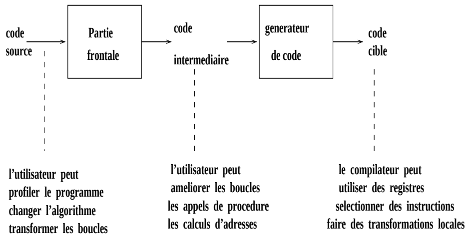
points chauds d'un programme : portion de code responsable d'à peu près 90% du temps d'exécution.

En pratique, on améliore les boucles les plus internes, d'où la nécessité d'une analyse globale de flot de données.

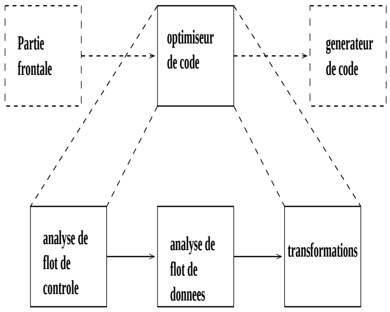
Critères pour optimiser le code : bénéfice maximal pour un effort minimal.

1. Une transformation doit préserver la sémantique d'un programme. Elle ne doit pas changer le résultat produit par un programme pour une entrée donnée, ni produire une erreur qui n'était pas présente dans la version d'origine.
2. une transformation doit en moyenne accélérer les programmes d'une quantité mesurable.
3. une transformation doit mériter l'effort qu'on lui consacre. Certaines optimisations locales (ou *à lucarne*) sont suffisamment simples et avantageuses pour être incluses dans chaque compilateur.
4. C'est uniquement lorsque le programme occupe une fraction significative des cycles de la machine que l'amélioration de la qualité du code justifie le temps passé à l'optimisation.

Place des tâches d'optimisation : en améliorant le programme à tous les niveaux, on peut obtenir des améliorations spectaculaires du temps d'exécution de ce programme (on peut passer de quelques heures à quelques secondes).



Comment optimiser en transformant le code intermédiaire ?



Dans la suite, on illustre sur l'exemple suivant :

```
void TriRapide(m, n)
{
    int i, j;
    int v, x;
    if (n <= m) return;
    i = m-1; j=n; v=a[n];                                (*)
    while ( 1 ){
       do i = i+1; while (a[i] < v);
       do j = j-1; while (a[j] > v);
       if (i >= j) break;
       x = a[i]; a[i] = a[j]; a[j] = x;
    }
    x = a[i]; a[i] = a[n]; a[n] = x;                     (**)
    TriRapide(m, j); TriRapide(i+1, n);
}
```


On traduit le programme de l'instruction (*) à l'instruction (**) :

(1) i := m-1
(2) j := n
(3) t_1 := 4*n
(4) v := a[t_1]
(5) i := i + 1
(6) t_2 := 4*i
(7) t_3 := a[t_2]
(8) si t_3 < v aller à (5)
(9) j := j-1
(10) t_4 := 4*j
(11) t_5 := a[t_4]
(12) si t_5 > v aller à (9)
(13) si i >= j aller à (23)
(14) t_6 := 4*i
(15) x := a[t_6]
(16) t_7 := 4*i
(17) t_8 := 4*j
(18) t_9 := a[t_8]
(19) a[t_7] := a[t_9]
(20) t_{10} := 4*j
(21) a[t_{10}] := x
(22) aller à (5)
(23) t_{11} := 4*i
(24) x := a[t_{11}]
(25) t_{12} := 4*i
(26) t_{13} := 4*n
(27) t_{14} := a[t_{13}]
(28) a[t_{12}] :=t_{14}
(29) t_{15} := 4*n
(30) a[t_{15}] := x

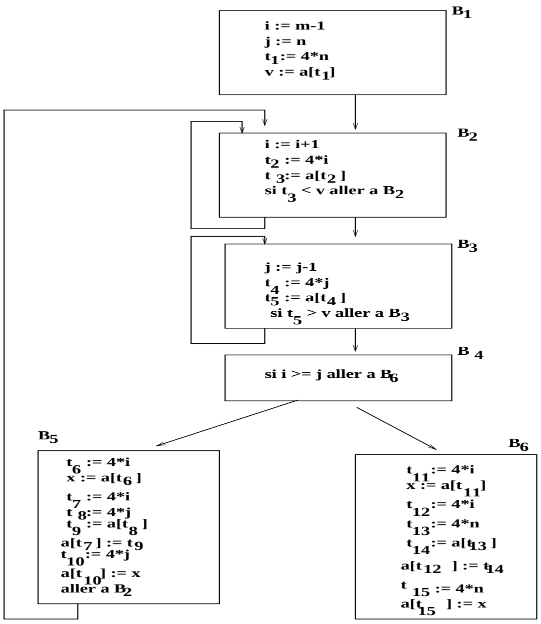
6.2 Les principales sources d'optimisation.

Transformation locale : qui peut être effectuée en examinant uniquement les instructions d'un bloc de base.
Par opposition aux transformations globales.

Beaucoup de transformations peuvent être effectuées à la fois au niveau local et au niveau global.

En général, on effectue d'abord les transformations locales.

Graphe de flot de contrôle du code :



(a) Sous-expressions communes.

Une occurrence d'une expression e est appelée *sous-expression commune* si e a été calculée précédemment et si les valeurs des variables apparaissant dans e n'ont pas été changées depuis le calcul précédent.

On élimine localement les sous-expressions communes :

```
t6 := 4*i
x := a[t6 ]
t7 := 4*i
t8 := 4*j
t9 := a[t8 ]
a[t7] := t9
t10 := 4*j
a[t10] := x
aller a B2
```

B_5 avant

```
t6 := 4*i
x := a[t6]
t8 := 4*j
t9 := a[t8]
a[t6] := t9
a[t8] := x
aller a B2
```

B_5 apres

(a) Sous-expressions communes.

Autre élimination :

```
t11 := 4 * i
x := a[t11]
t12 := 4*i
t13 := 4*n
t14 := a[t13]
a[t12] := t14
t15 := 4*n
a[t15] := x
```

B_6 avant

```
t11 := 4 * i
x := a[t11]
t13 := 4*n
t14 := a[t13]
a[t11] := t14
a[t13] := x
```

B_6 apres

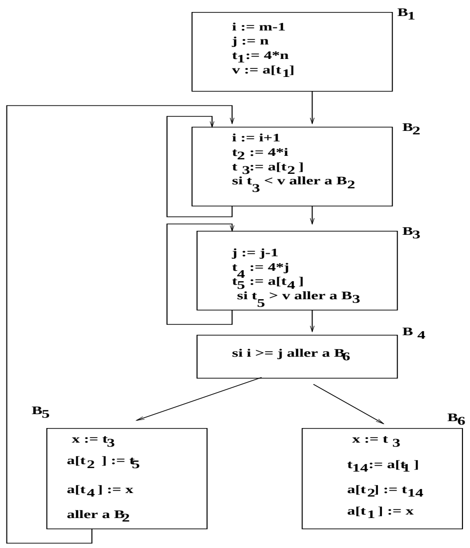
(a) Sous-expressions communes.

On élimine globalement les sous-expressions communes :
Dans B_5, la séquence
  t_6 := 4*i
  x := a[t_6]
est remplacée par x := t_3.
En effet, lorsque le contrôle passe de l'évaluation de 4*i dans B_3 à B_5, i n'est pas modifié. t_3 peut donc être utilisé si on a besoin de a[4*i].

De la même façon, la séquence
  t_8 := 4*j
  t_9 := a[t_8]
  a[t_6] := t_9
  a[t_8] := x
peut être remplacée par
  a[t_2] := t_5
  a[t_4] := x

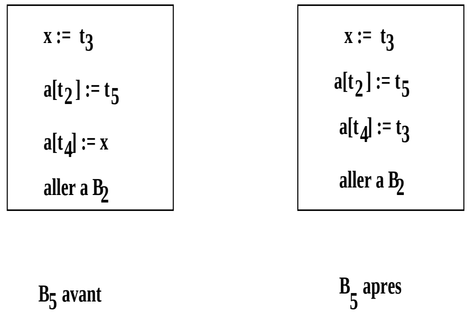
(a) Sous-expressions communes.

En appliquant l'élimination des sous-expressions de façon globale, on obtient le graphe de flot de contrôle suivant :



(b) Propagation des copies.

Idée : la transformation par propagation des copies consiste à utiliser g à la place de f, chaque fois que c'est possible après l'instruction de copie f := g. Par exemple, B_5 devient :

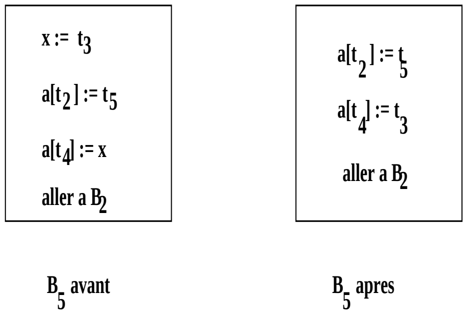


Conséquence : ça peut donner la possibilité d'éliminer les affectations à x.

Propagation des constantes : remplacer une expression constante par sa valeur. Exemple :
remplacer 2 * 3.14 par 6.28

(c) Elimination du code inutile.

code inutile = instructions qui calculent des valeurs qui ne sont jamais utilisées.
Le code inutile peut apparaître comme résultat d'une transformation.



(d) Optimisation de boucles.

3 techniques importantes :

1. déplacement de code : transfère du code à l'extérieur des boucles.
2. élimination des variables d'induction.
3. réduction de force : remplace une opération coûteuse par une opération de coût moins élevé. Exemple : remplacer une multiplication par une addition.

(d) Optimisation de boucles.

Définition. Boucle :
C'est un ensemble de nœuds d'un gfc tel que

1. tous les nœuds de la collection sont fortement connectés, i.e. il existe un chemin de longueur 1 ou plus, joignant chaque nœud de la boucle à n'importe quel autre nœud, entièrement situé à l'intérieur de la boucle et,
2. la collection de nœuds a une entrée unique, i.e. la seule façon d'atteindre un nœud quelconque de la boucle à partir d'un nœud situé à l'extérieur est de passer par cette entrée.

Une boucle est dite interne si elle ne contient pas d'autre boucle.

En général, les boucles sont traitées de l'intérieur vers l'extérieur.

(d) Optimisation de boucles.

Déplacement de code : prendre une expression qui produit le même résultat pour toutes les itérations et déplacer cette expression devant la boucle. (calcul d'un invariant de boucle).

Exemple :

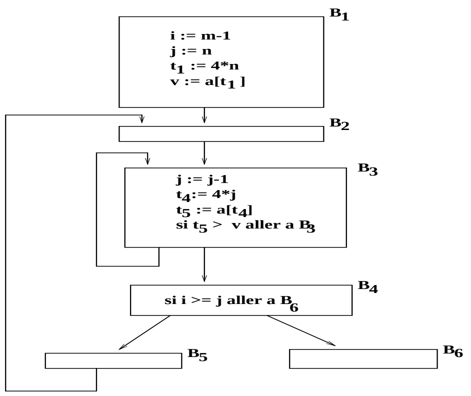
```
tant que (i <= limite - 2)
    .... (aucune instruction ne change la valeur de limite)
```

est remplacée par

```
t := limite - 2;
tant que (i <= t)
    ....
```


(d) Optimisation de boucles.

Variables d'induction et réduction de force.
Considérons le bloc B_3 :



On remarque que les valeurs de j et t_4 sont couplées. Chaque fois que j décroît de 1, t_4 diminue de 4. j et t_4 sont des variables d'induction.

Quand il y a au moins 2 variables d'induction dans une boucle, on peut les éliminer toutes sauf une.

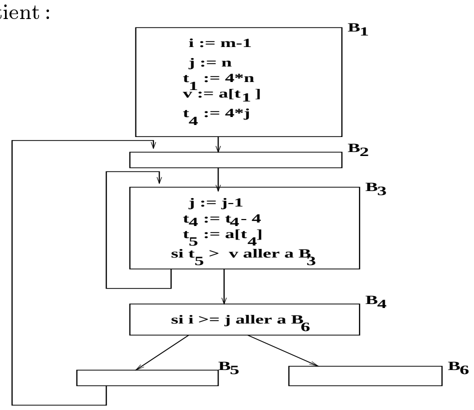
Définition : Variable d'induction.
x est une variable d'induction d'une boucle si, chaque fois que x change de valeur, elle est incrémentée ou décrémentée par une certaine constante.

(d) Optimisation de boucles.

t_4 := 4*j est remplacée par t_4 := t_4 - 4.
A l'entrée dans B_3, t_4 n'a pas de valeur, Cela nous amène à ajouter l'instruction t_4 := 4*j à la fin de B_1.
Avantages de cette transformation :

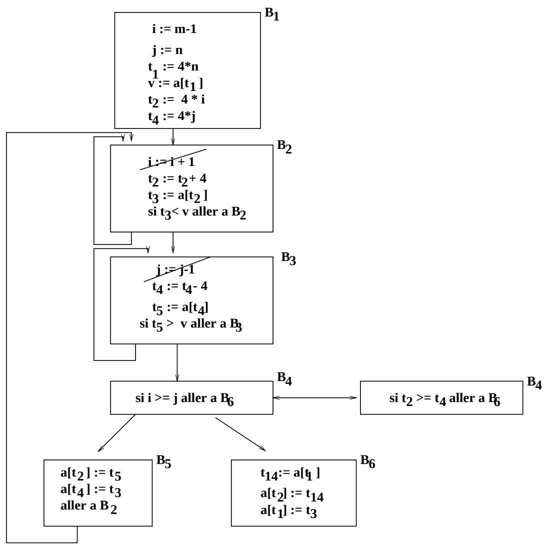
- remplacer la multiplication par une soustraction ;
- une seule multiplication à l'extérieur de la boucle B_3-B_3.

On obtient :



(d) Optimisation de boucles.

On peut faire la même chose pour B_2. On obtient le gfc suivant après élimination des variables d'induction.



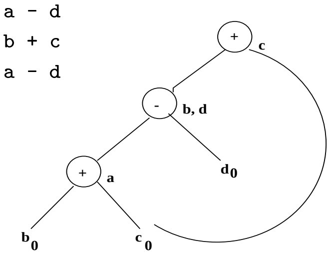
(e) Techniques implantées pour effectuer les transformations.

Pour l'optimisation des blocs de base :

1. La détection des sous-expressions communes se fait en construisant un DAG pour chaque bloc de base.

Exemple :

```
a := b + c
b := a - d
c := b + c
d := a - d
```



Si b (resp. d) inactive à la sortie du bloc,

```
a := b + c
d := a - d  (resp. b := a - d)
c := b + c
```

si b et d sont actives à la sortie du bloc

```
a := b + c
b := a - d
c := b + c
d := b
```


(e) Techniques implantées pour effectuer les transformations.

2. Elimination du code inutile : en utilisant également un DAG. Il suffit de supprimer toutes les racines qui n'ont pas de variable active.

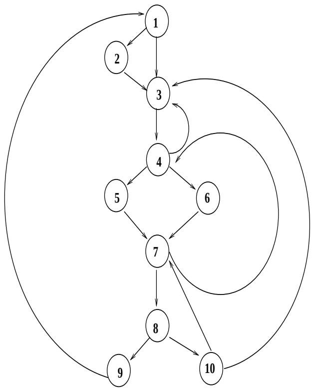
3. Optimisation des boucles : par une technique de calcul des *dominants* et un algorithme qui vérifie qu'un gfc est *réductible*.

Définition : nœud dominant.
Un nœud d d'un gfc domine le nœud n si tout chemin partant du nœud initial du gfc et arrivant à n passe par d.

Remarques :

- tout nœud se domine lui-même.
- l'entrée d'une boucle domine tous les nœuds de la boucle.

Exemple : considérons le fgc suivant dont le nœud initial est 1 :

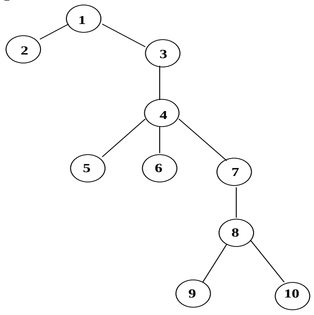


(e) Techniques implantées pour effectuer les transformations.

On représente l'information des dominants par un arbre :

Définition : Arbre des dominants.
Sa racine est le nœud initial, et chaque nœud domine uniquement ses descendants dans l'arbre.

Exemple :



Pour déterminer les boucles, on s'appuie sur leurs deux propriétés suivantes :

1. une boucle doit avoir un point d'entrée unique appelé *en-tête*.
   Ce point d'entrée doit dominer tous les autres nœuds de la boucle (sinon, il ne serait pas la seule entrée de la boucle).
2. il doit exister au moins une façon de répéter la boucle, c'est- à-dire un chemin qui revient vers l'en-tête.

(e) Techniques implantées pour effectuer les transformations.

On détermine les boucles d'un gfc en recherchant les arcs du gfc dont l'arrivée domine le départ.
Exemple : il y a un arc $7 \to 4$ et 4 domine 7. De même, il y a un arc $7 \to 10$ et 7 domine 10.
Même chose pour $4 \to 3$, $8 \to 3$ et $9 \to 1$.

Définition : Arcs de retour.
Ce sont tous les arcs $n \to d$ tels que d domine n.

Définition : boucle naturelle.
Soit $n \to d$ un arc de retour. La boucle naturelle de l'arc contient d et l'ensemble des nœuds qui peuvent atteindre n sans passer par d.
d est l'en-tête de la boucle.
Exemple : la boucle naturelle de l'arc $10 \to 7$ est formée des nœuds 7, 8 et 10.
La boucle naturelle de l'arc $9 \to 1$ est formée de tous les nœuds du graphe.

(e) Techniques implantées pour effectuer les transformations.

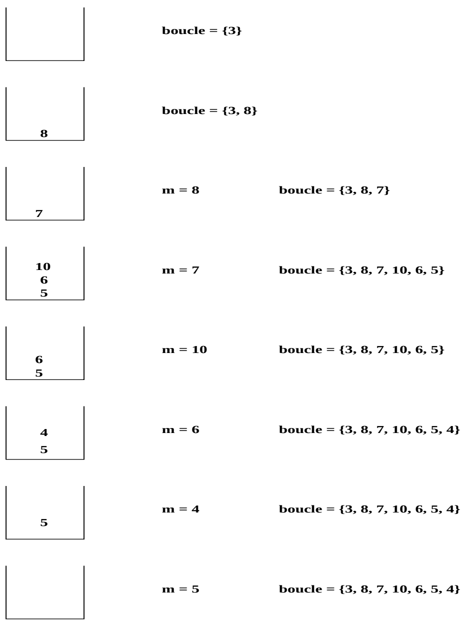
Algorithme de calcul d'une boucle naturelle :
Données : un gfc G et un arc de retour $n \to d$.
Résultat : l'ensemble *boucle* contenant tous les nœuds de la boucle naturelle de $n \to d$.

procédure Insérer(m) ;
  si m n'est pas dans *boucle* **alors**
    boucle := *boucle* ∪ {m}
    empiler(m, p)
  fsi
fin.

le programme principal :
début
pile := vide ;
boucle := {d}
Insérer(n) ;
tant que pile non vide **faire**
  m := sommet(p) ; dépiler(p) ;
  pour chaque prédécesseur p de m **faire**
    Insérer(p) ;
ftq
fin.

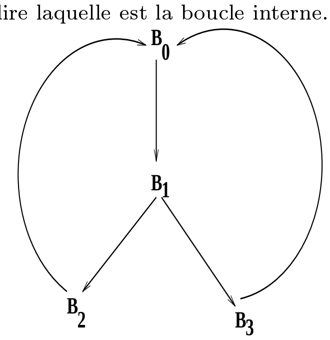
(e) Techniques implantées pour effectuer les transformations.

Exemple : calcul de la boucle naturelle de l'arc $8 \to 3$:



(e) Techniques implantées pour effectuer les transformations.

La notion de boucle naturelle coïncide avec celle de boucle interne (boucle qui ne contient pas d'autre boucle).
Problème : quand deux boucles ont la même en-tête, il est difficile de dire laquelle est la boucle interne.



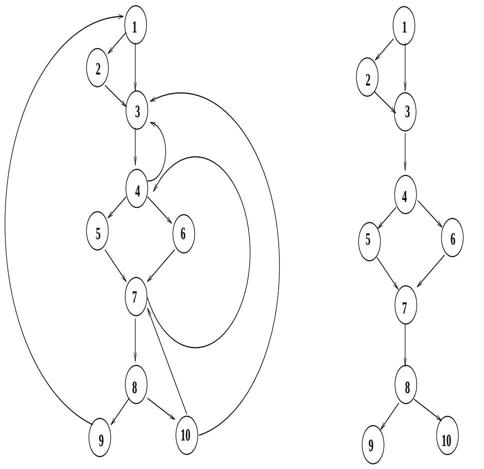
Définition : gfc réductible
Un gfc est réductible s'il est possible de partitionner les arcs en 2 groupes disjoints, appelés *arcs en avant* et *arcs en arrière*, possédant les propriétés suivantes :

- les arcs en avant forment un graphe acyclique dans lequel chaque nœud peut être atteint par le nœud initial de G ;
- les arcs en arrière sont formés uniquement des arcs dont l'arrivée domine le départ.

(e) Techniques implantées pour effectuer les transformations.

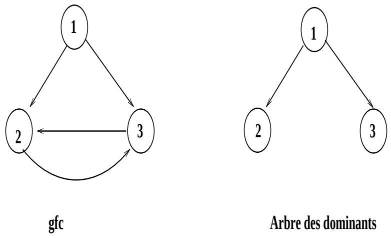
En pratique, les gfc que l'on rencontre sont réductibles.
Dans les langages structurés, les instructions produisent des gfc réductibles.
Si on a l'arbre des dominants d'un gfc, on peut déterminer et supprimer tous les arcs en arrière. Si le graphe est réductible, les arcs restants doivent être des arcs en avant et le test de réductibilité revient à tester que les arcs en avant forment un graphe acyclique.

Exemple : si on supprime tous les arcs en arrière, le gfc restant est acyclique.



(e) Techniques implantées pour effectuer les transformations.

Le gfc suivant n'est pas réductible. En effet, aucun arc en arrière n'existe. Il serait réductible si le graphe entier était acyclique!



6.3 Analyse globale de flot de données.

Pour optimiser le code, le compilateur doit être capable de rassembler des informations sur le programme dans son ensemble et de communiquer ces informations à chaque bloc du gfc.
Exemples :

- connaître à la fin de chaque bloc, les variables qui sont référencées plus tard permet d'améliorer l'utilisation des registres.
- connaître les sous-expressions communes globales permet d'éliminer des calculs redondants.

Les informations de flot de données sont rassemblées par la mise en place et la résolution de systèmes d'équations qui relient les informations en divers points du programme. Une équation a la forme générale :

$$\text{Out(I)} = \text{Prod(I)} \cup (\text{In(I)} - \text{Supp(I)})$$

qui se lit : les informations valides à la fin d'une instruction sont produites à l'intérieur de cette instruction, ou alors elles entrent au début et elles ne sont pas supprimées au cours de l'instruction.

De telles équations sont appelées équations de flot de données.
En général, les équations sont mises en place au niveau des blocs de base plutôt qu'au niveau des instructions puisque les blocs ont un point de sortie unique.

Certaines instructions mettent en jeu des subtilités, comme l'appel de procédure, l'affectation au travers d'un pointeur.

A l'intérieur d'un bloc de base, on parlera du point entre 2 instructions, du point situé avant la 1ère instruction, et du point après la dernière.

B_1

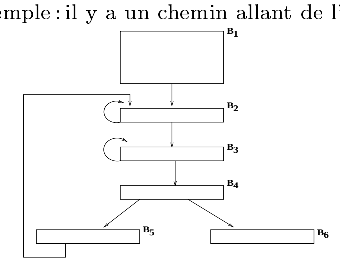
d_1: i := m - 1
d_2: j := n
d_3: a := u_1

Exemple : le bloc suivant a 4 points :

Globalement, si on considère tous les points de tous les blocs, un chemin de p_1 à p_n est une suite de points $p_1, p_2, \ldots,$ p_n telle que, $\forall\ i,\ 1 \leq i \leq n$,

- soit p_i est le point qui précède immédiatement une instruction et p_{i+1} est le point qui suit immédiatement cette instruction dans le même bloc ;
- soit p_i est la fin d'un bloc et p_{i+1} est le début d'un bloc successeur.

Exemple : il y a un chemin allant de l'entrée de B_5 à l'entrée de B_6.



Définition d'une variable.

Une définition d'une variable x est une instruction qui affecte ou peut affecter une valeur à x.

- définitions non ambiguës : affectation, instruction de lecture.
- définitions ambiguës :
  - appel de procédure avec x pour argument lors d'un passage par adresse, ou une procédure qui peut accéder à x parce que x est dans sa portée.
  - une affectation au travers d'un pointeur qui pourrait désigner x.

Définition visible.

Une définition d est visible depuis un point p s'il existe un chemin allant du point suivant immédiatement d jusqu'à p tel que d ne soit pas supprimée le long de ce chemin.

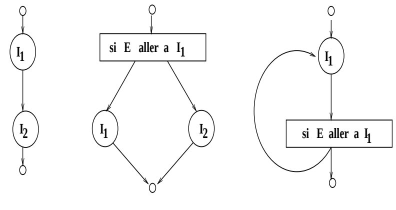
Une définition de a est supprimée entre deux points si entre ces deux points, il y a une nouvelle définition non ambiguë de a.

Nous donnons une définition de In, Out, Prod, Supp pour calculer en tout point d'un programme l'ensemble des définitions visibles.

d $\in$ Prod(I) si d est visible à la sortie de I (visible ou non à l'entrée).

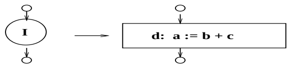
d $\in$ Supp(I) si d n'est jamais visible à la sortie de I.

Quelques constructions de contrôle structurées :

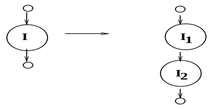


si E alors I_1 sinon I_2 faire I_1 tant que E

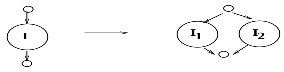
Equations de flot de données pour les définitions visibles :



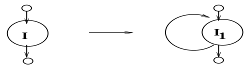
prod(I) = {d}
Supp(I) = D_a - {d}
Out(I) = Prod(I) U (In(I) - Supp(I))



Prod(I) = Prod(I_2) U (Prod(I_1) - Supp(I_2))
Supp(I) = Supp(I_2) U (Supp(I_1) - Prod(I_2))
In(I_1) = In(I)
In(I_2) = Out(I_1)
Out(I) = Out(I_2)



Prod(I) = Prod(I_1) U Prod(I_2)
Supp(I) = Supp(I_1) Supp(I_2)
In(I_1) = In (I) In(I_2) = In(I)
Out(I) = Out(I_1) U Out(I_2)



Prod(I) = Prod(I_1)
Supp(I) = Supp(I_1)
In(I_1) = In(I) U Prod(I_1)
Out(I) = Out(I_1)

Au départ, In(programme complet) = $\emptyset$

Remarques :

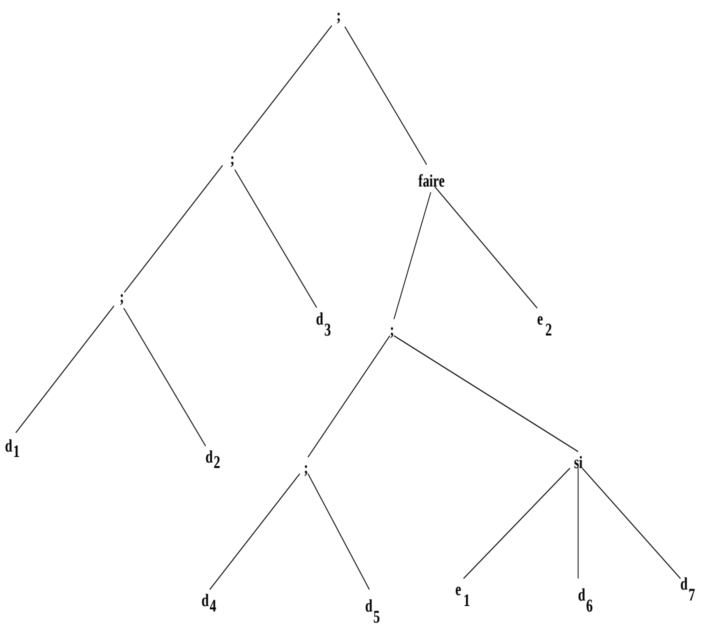
- le Prod véritable est toujours un sous-ensemble du Prod calculé. surestimer Prod : empêche une optimisation mais n'altère pas le sens du code.
  Sous-estimer Prod serait une erreur. On pourrait remplacer une variable par sa valeur en un point alors que ce n'est pas toujours la seule définition valide.
- le Supp véritable est toujours un sur-ensemble du Supp calculé.

L'évaluation du système d'équations de flot se fait en 2 passes :

1. on synthétise Prod et Supp.
2. on hérite In et on synthétise Out.

Exemple :

d_1 : i := m - 1
d_2 : j := n
d_3 : a := u_1
faire
d_4 : i := i + 1
d_5 : j := j - 1
si e_1 alors
d_6 : a := u_2
sinon
d_7 : i := u_3
tant que e_2



Cette méthode est simple et efficace lorsqu'elle peut s'appliquer. Malheureusement, elle n'est pas assez générale pour des langages comme Pascal qui permettent des gfc quelconques.

Une autre méthode : au lieu d'utiliser l'arbre syntaxique pour calculer les In et les Out, on construit d'abord le gfc et on calcule simultanément In et Out pour chaque nœud. Les équations sont les mêmes que les précédentes à deux différences près :

1. les In sont calculés à partir des Out (le problème est exprimé par des équations arrière).
2. lorsque plus d'un arc arrive à l'entrée de B, l'ensemble des définitions visibles depuis l'entrée de B est l'union des définitions qui arrivent par chacun des arcs.
   En revanche, l'ensemble des expressions disponibles à l'entrée de B est l'intersection des expressions disponibles de chacun des arcs.

Définition : expression disponible.

e est disponible en un point p si tous les chemins depuis le nœud initial jusqu'à p évaluent e et si, après la dernière de ces évaluations sur chacun de ces chemins, il n'y a d'affectation à aucun des opérandes de e.

(a) Equations

Algorithme itératif pour le calcul des définitions visibles.
A l'intérieur de chaque bloc, on calcule Prod et Supp comme précédemment, puis on considère chaque bloc comme une instruction :

$$\mathrm{In(B)} = \bigcup_{p \in \mathrm{P(B)}} \mathrm{out(B)}$$

P(B) est l'ensemble des blocs prédécesseurs de B dans le gfc.

$$\mathrm{Out(B)} = \mathrm{Prod(B)} \ \cup \ (\mathrm{In(B)} - \mathrm{Supp(B)})$$

On utilise une méthode itérative partant de $\mathrm{In(B)} = \emptyset$. Pour un gfc qui comporte n blocs, on obtient 2n équations qu'on résoud itérativement jusqu'à convergence du résultat.

L'algorithme :

Données : un gfc pour lequel Supp(B) et Prod(B) ont été calculés pour tout B.

Résultat : les ensembles In(B) et Out(B) pour chaque B.

Initialiser In(B) à $\emptyset$ pour chaque bloc B

pour chaque bloc B **faire**

  Out(B) = Prod(B) ;

répéter

  convergence := **vrai**

  pour chaque bloc B **faire**

    In(B) := $\bigcup_{\mathrm{p} \in \mathrm{P(B)}} \mathrm{Out(p)}$

    ancien_out := out(B)

    Out(B) := Prod(B) $\cup$ (In(B) - Supp(B)) ;

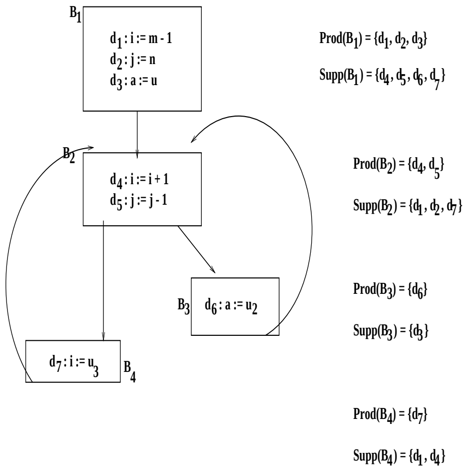
    si Out(B) $\neq$ ancien_out **alors**

      convergence := **faux**

  fin

jusqu'à convergence.

Exemple :



Bloc B	In(B) initiaux	Out(B)	In(B) Passe 1	Out(B)
B_1	$\emptyset$	$\{d_1\text{-}d_3\}$	$\emptyset$	$\{d_1\text{-}d_3\}$
B_2	$\emptyset$	$\{d_4,\ d_5\}$	$\{d_1\text{-}d_3, d_6, d_7\}$	$\{d_3\text{-}d_6\}$
B_3	$\emptyset$	$\{d_6\}$	$\{d_3\text{-}d_6\}$	$\{d_4\text{-}d_6\}$
B_4	$\emptyset$	$\{d_7\}$	$\{d_3\text{-}d_6\}$	$\{d_3,\ d_5\text{-}d_7\}$

Bloc B	In(B) Passe 2	Out(B)
B_1	$\emptyset$	$\{d_1\text{-}d_3\}$
B_2	$\{d_1\text{-}d_7\}$	$\{d_3\text{-}d_6\}$
B_3	$\{d_3\text{-}d_6\}$	$\{d_4\text{-}d_6\}$
B_4	$\{d_3\text{-}d_6\}$	$\{d_3, d_5\text{-}d_7\}$

(b) Réordonnancement des instructions.

Pour les processeurs VLIW et pour les processeurs pipelinés, les séquences d'instructions générées doivent être réordonnées pour utiliser au mieux le pipeline ou les unités susceptibles de travailler en parallèle.

On réordonne les instructions d'un bloc de base. Pour cela, on tient compte des dépendances entre les instructions. Les dépendances sont déterminées par les accès successifs (en lecture ou en écriture) à des composantes de l'état de la machine telles que les cellules mémoire, les registres.

Ces modifications d'état sont appelées **définitions**

$$X := \text{définition d'une composante } X$$

ou **utilisations**

$$:= X \text{ une utilisation de } X$$

Remarque : une instruction peut utiliser ou définir plusieurs ressources.

(b) Réordonnancement des instructions.

Exemple :

```
a: X :=
b: X :=
c:   := X
d: X :=
```

Si on échange les instructions a et b, on changerait la valeur fournie à l'utilisation dans c. Même remarque si on échange b et c, ou si on échange c et d.

Types de définitions :

- modification du contenu d'un registre par une instruction de chargement, par le rangement du résultat d'une opération, ou par une opération de décrémentation ou d'incrémentation ;
- la modification des bits du code de condition, indiquant un débordement, une retenue ou le résultat d'une comparaison ;
- l'écriture de valeurs dans des cellules mémoire ;
- la modification d'un pointeur de pile par des instructions de gestion de procédures.

(b) Réordonnancement des instructions.

Types d'utilisations :

- l'utilisation du contenu d'un registre pour une opération ou un calcul d'adresse ;
- la recopie d'un registre en mémoire ;
- le chargement du contenu d'une cellule mémoire ;
- le test d'une partie du code de condition dans une instruction de branchement conditionnel ;

Par conséquent, différentes sortes de dépendances entre instructions qui imposeront une exécution séquentielle :

- **définition-définition (d-d)** comme entre les instructions a et b.
  b parait annuler l'effet de a et rendre a superflue.
  Attention ! b pourrait contenir une utilisation de X.
- **définition-utilisation (d-u)** comme entre b et c.
- **utilisation-définition (u-d)** comme entre c et d.

(b) Réordonnancement des instructions.

Problème : comment extraire précisément les dépendances d'un bloc de base ?

Difficulté : deux adressages différents peuvent désigner la même cellule mémoire.

1ère approche : considérer l'ensemble de la mémoire comme un seul objet

$\Rightarrow$ deux ressources de la machine : la mémoire et les registres.

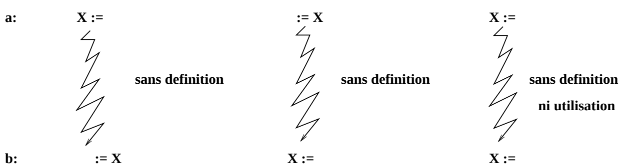
On propose une définition de graphe de dépendances directes.

On obtient **toutes** les dépendances en effectuant la fermeture transitive de ce graphe.

(b) Réordonnancement des instructions.

Définition. Graphe de dépendances d'un bloc de base
Soit un bloc de base d'un programme en langage machine. Son graphe de dépendances est le graphe orienté, étiqueté et acyclique, défini par :

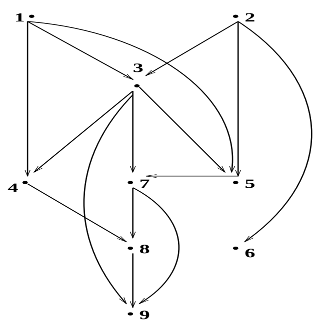
- les nœuds sont étiquetés par les instructions du bloc ;
- un arc joint le nœud de a au nœud de b si a se trouve avant b dans la séquence d'instructions, et si une des conditions suivantes est satisfaite :
  1. a définit une ressource que b utilise et le chemin de a vers b ne contient pas de définition de cette ressource ;
  2. a utilise une ressource que b définit, et le chemin de a vers b ne contient pas de définition de cette ressource ;
  3. a et b définissent la même ressource, et le chemin de a vers b ne contient ni définition ni utilisation de cette ressource.



(b) Réordonnancement des instructions.

Exemple : un bloc de base et son graphe de dépendances.

```
1:  D1      := M[A1+4];
2:  D2      := M[A1+6];
3:  A1      := A1+2;
4:  D1      := D1 + A1;
5:  M[A1]   := A1;
6:  D2      := D2+1;
7:  D3      := M[A1 + 12];
8:  D3      := D3 + D1;
9:  M[A1+6] := D3
```



Exemple : on considère le nœud 9. L'arc $3 \rightarrow 9$ existe parce que 3 définit A1 et 9 l'utilise.

l'arc $7 \rightarrow 9$ existe en raison du conflit **u-d** sur M.

arc $8 \rightarrow 9$ existe parce que 8 définit D3 et 9 l'utilise.

(b) Réordonnancement des instructions.

Algorithme de construction du graphe de dépendances.

Donnée : un bloc de base B
Résultat : le graphe de dépendances de B
fonction conflit(ressource, inst_1, inst_2) : type de conflit.
détermine s'il y a un conflit entre les deux instructions pour ressource et retourne le type de conflit, s'il y en a un.
procedure CréerArc(a $\rightarrow$ b)
crée un arc de a vers b s'il n'en existe pas déjà un.

Remarques :

- Les définitions et utilisations sont sous forme de couples (ressource, occurrence d'instruction) ;
- l'ensemble DerDef contient les dernières définitions de ressources traitées,
- l'ensemble UtilExpo contient les utilisations déjà traitées, et qui ne sont pas réécrites par des définitions ultérieures.
- l'algorithme parcout le bloc en partant de la fin.

Idée : tester si les définitions et les utilisations de l'instruction courante se trouvent en dépendance avec les instructions dans DerDef ou UtilExpo.

(b) Réordonnancement des instructions.

Début
Inst := dernière instruction du bloc ;
CréerNœud(Inst) ;
DerDef := {(r, Inst) | $r \in$ Définitions(Inst)} ;
UtilExpo := {(r, Inst) | $r \in$ Utilisations(Inst)} ;
tant que prédécesseur(Inst) est défini **faire**
Inst := prédécesseur(Inst) ;
pour toute ressource r, définie ou utilisée dans Inst **faire**
pour tous (r,b) $\in$ DerDef $\cup$ UtilExpo **faire**
cas conflit(r, Inst, b) **est**
d-d :
si il n'existe aucun couple (r, α) $\in$ UtilExpo
alors CréerArc(Inst $\rightarrow$ b) ;
d-u ou **u-d** :
CréerArc(Inst $\rightarrow$ b) ;
fin cas ;
fait
fait ;
pour toute ressource r' définie dans Inst **faire**
DerDef := DerDef - {s $\in$ DerDef| $\exists\alpha$: s=(r',α)} $\cup$ {(r',Inst)} ;
UtilExpo := UtilExpo - {b $\in$ UtilExpo| $\exists\beta$: b = (r',β)} ;
fait ;
pour toute ressource r' utilisée dans Inst **faire**
UtilExpo := UtilExpo $\cup$ {(r', Inst)} ;
fait ;
fait
fin.

(b) Réordonnancement des instructions.

Le graphe de dépendances représente les différents degrés de liberté laissés pour réordonner les instructions. Tout tri topologique du graphe doit donner un ordre admissible.

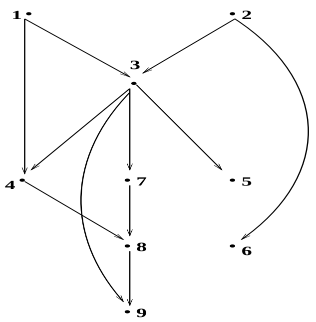
Moins le graphe de dépendances a d'arcs, plus il offre de possibilités pour l'ordonnancement.

2ème approche : on réduit le nombre d'arcs du graphe de dépendances en différenciant les cellules mémoire adressées : **Analyse d'alias**.

C'est facile lorsque toutes les cellules mémoire sont adressées par le même registre. Les adresses des différentes cellules d'un bloc de base ne se différencient que par des constantes. On peut donc les distinguer facilement.
$\Rightarrow$ une cellule mémoire est considérée comme une ressource.

(b) Réordonnancement des instructions.

Exemple : une analyse d'alias sur l'exemple précédent permet de supprimer certains arcs.



(b) Réordonnancement des instructions.

Hélas, il peut exister d'autres restrictions sur l'ordre des instructions comme par exemple, celle qui impose un délai minimal v entre deux instructions a et b en dépendance. Elle impose d'ordonner a et b de telle façon que leurs exécutions soient séparées par au moins v cycles.

Les pipelines :
Hypothèse : une machine cible qui est un processeur avec un pipeline d'instructions à k étages.
But : réordonner les séquences d'instructions dans les blocs de base, pour obtenir les séquences les plus courtes possibles.

La technique est fondée sur le tri topologique du graphe de dépendances.

Le tri topologique qui transforme un ordre partiel en un ordre total commence par les éléments minimaux. Lorsqu'il est amené à choisir un élément pour lui affecter un rang, il choisit une instruction qui ne présente aucune collision avec celles déjà ordonnées. S'il n'en existe pas, une instruction NOP est insérée au code.

7. Influence des nouvelles architectures sur la production de compilateurs :

1. Introduction
2. Les processeurs CISC
3. L'évolution technologique
4. Les processeurs RISC
5. Le processeur SPARC de SUN Microsystems

7.1 Introduction

L'objectif à atteindre lors de la conception d'un nouveau processeur est de réduire le fossé sémantique entre les langages de haut niveau et le jeu d'instructions du processeur,
i.e. réduire la différence de puissance d'expression entre le bas niveau (représenté par le code machine) et le haut niveau (représenté par le langage de programmation dit évolué) en gardant compatibles les codes exécutables.

Relation avec les langages de programmation
L'adéquation à un large éventail de langages de programmation traduit l'influence des logiciels et en particulier des compilateurs dans la conception d'une architecture.
Objectif : obtenir une machine générale.

On peut voir le problème de façon inverse : l'évolution technologique a donné naissance à de nouvelles architectures ⇒
influence sur la production de compilateurs.

7.2 Les processeurs CISC

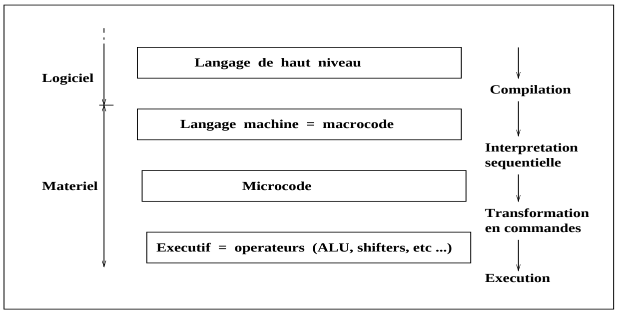
Postulat : la performance d'un processeur provient en grande partie de l'exécution d'instructions complexes de haut niveau par le matériel et de son corollaire, i.e. la réduction du nombre d'instructions à exécuter pour une même tâche.

Comment se mesure la performance d'un processeur ?
directement liée au temps nécessaire à l'exécution d'une tâche donnée.
3 critères :

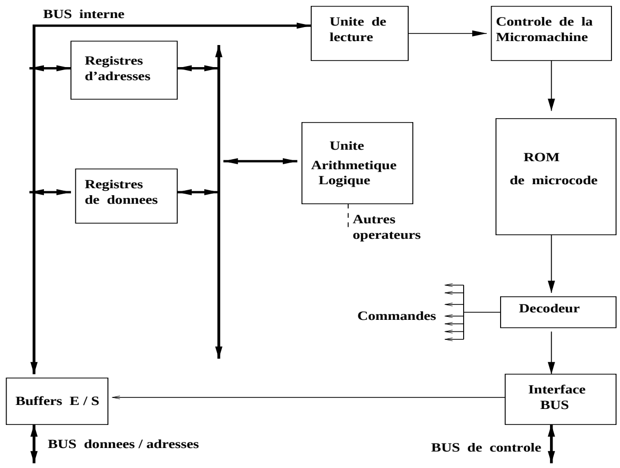
1. nombre d'instructions nécessaires à l'exécution de la tâche.
2. nombre moyen de cycles machine nécessaire à l'exécution d'une instruction.
3. durée de chaque cycle machine.

Développement d'une architecture CISC
Principe : relative indépendance entre la machine réelle et la machine logique vue de l'utilisateur.
Comment ? par un modèle d'exécution à plusieurs niveaux qui, par le biais de la microprogrammation, émule le modèle de machine souhaité sur une micromachine constituée d'un interpréteur et d'opérateurs élémentaires.

Schéma d'exécution d'un processeur CISC :



Structure générale d'un processeur CISC :



7.3 L'évolution technologique :

Le contexte technologique évolue :

1. l'évolution des mémoires intégrées tend à diminuer l'intérêt d'une mémoire interne de microprogrammes dont le temps d'accès devient sensiblement identique à ceux des composants mémoires externes.
2. l'avènement des techniques de mémoire cache.
3. les progrès réalisés en CAO électronique et au niveau des filières technologiques ⇒ de nouvelles possibilités aux concepteurs de systèmes.
4. le nombre très important de logiciels disponibles permet maintenant de mieux analyser le comportement des compilateurs
   ⇒ maturité des techniques de compilation.
   L'expérience montre indiscutablement qu'un ensemble d'instructions simples et uniformes est plus adapté.

Il est clair en effet que plus le jeu d'instructions est complexe, plus la conception d'un compilateur est longue et délicate.
En pratique, il est pratiquement impossible d'exploiter pleinement les potentialités offertes par un jeu d'instructions complexe, et pour chaque instruction, de sélectionner le meilleur mode d'adressage !

Ceci est conforté par divers travaux ayant pour objectif d'établir un jeu d'instructions unique. On distingue 8 classes d'instructions :

1. pour la manipulation de données.
2. de modification du séquencement du programme.
3. instructions arithmétiques.
4. instructions de comparaison.
5. instructions logiques.
6. instructions de décalage.
7. de manipulation de bits.
8. instructions d'E/S et diverses.

Sur le MC68000 de Motorola, des statistiques ont été établies pour calculer les occurrences d'utilisation du jeu d'instructions sur l'ensemble des 8 classes. Les résultats ont montré que 31 instructions sur les 76 que comporte le processeur MC68000 auraient pu être éliminées sans remettre fondamentalement en question les performances du processeur.

⇒ remise en question des processeurs CISC.

7.4 Les processeurs RISC :

Un principe fondamental est d'intégrer (sous la forme de circuits logiques) des fonctionnalités à forte occurrence et seulement celles-ci.

Le projet RISC-I : 4 contraintes principales sur la méthodologie de conception :

1. exécuter une instruction par cycle.
2. toutes les instructions ont une même taille.
3. les accès à la mémoire ne sont effectués qu'au travers de deux instructions : *load* et *store* ; les autres instructions opèrent uniquement dans les registres.
4. l'adéquation aux langages de haut niveau.

Les observations les plus importantes furent les suivantes :

- les instructions d'appel et de retour (*call* et *return*) sont celles qui exigent le plus de temps d'exécution.
- 80% des variables locales utilisées dans les programmes sont des scalaires.
- 90% des structures de données complexes sont des variables globales.
- la majorité des procédures n'échangent que 6 arguments au maximum.
- la profondeur maximale pour l'appel de procédure est en moyenne de 8, une profondeur plus grande arrivant dans moins de 1% des cas.

Conséquences :

1. mécanisme de fenêtres de registres pour accélérer les appels et retours de procédures.
2. mise en place de l'instruction de branchement retardé.
3. le temps d'exécution d'un cycle a été déterminé par le temps nécessaire pour lire 2 registres, effectuer une addition puis écrire le résultat dans un registre.

7.5 Le processeur SPARC de SUN Microsystems :

(Scalable Processor ARChitecture)
Repose sur une architecture relativement simple, qui doit lui permettre de s'adapter à la programmation en C et aux fonctionnalités de Unix. Une interface avec un coprocesseur flottant fait partie intégrante de l'architecture (pour les applications de calcul numérique).

Principales caractéristiques :

- 136 registres internes à 3 ports d'accès, 2 de lecture et 1 d'écriture organisés en 8 fenêtres à recouvrement partiel lors des appels fonctionnels.
- le pipeline d'exécution comprend 4 étages :
  1. le premier lit l'instruction
  2. le second décode les opérandes
  3. le troisième exécute les opérations sur l'UAL
  4. le dernier écrit le résultat dans le registre destination.
- 64 instructions dont la moitié s'exécute en un cycle machine. Les autres prennent entre 2 et 4 cycles selon la taille de la donnée à transmettre (octets, mots, doubles mots)
- 2 UAL (une pour les instructions, l'autre pour les calculs d'adresses).
- bus de données et bus d'adressage séparés.