Compilation

Frédérique Carrere

12 septembre 2013

Chapitre 1

Introduction

1.1 Qu'est-ce qu'un compilateur ?

1.1.1 Définition

Un compilateur est un logiciel qui traduuit un programme écrit dans un langage source vers un programme écrit dans un langage cible.
Langage L → Compilateur → Langage L'

D'ordinaire L est un langage de programmation de haut niveau, comme C++ ou Java par exemple, et L' est un langage de bas niveau, par exemple un code machine adapté à un ordinateur en particulier.

Exemples :
de C vers de l'assembleur : `gcc`
de Java vers du byte code : `javac`
de latex vers du postcript ou du html : `latex2html`

1.1.2 Rôle du compilateur

Son rôle est de permettre de programmer avec un haut niveau d'abstraction (ex : langages dédiés) plutôt qu'en assembleur. Il permet donc d'écrire des programmes lisibles, modulaires, maintenables, réutilisables.
Il permet de s'abstraire de l'architecture de la machine et donc d'écrire des programmes portables.
Il permet de détecter des erreurs et donc d'écrire des programmes plus fiables.
On peut écrire des compilateurs certifiés par des logiciels de vérifications

comme coq.

Ses principale tâches sont :

- lire et analyser le programme
- détecter des erreurs (lexicales, syntaxiques, sémantiques)
- écrire dans un langage intermédiaire la séquence d'instructions à effectuer par la machine
- optimiser cette séquence d'instructions (taille du code, vitesse d'éxecution)
- traduire ces instructions d'un langage intermédiaire dans le langage cible

1.1.3 Techniques mises en jeu

Le compilateur fait intervenir de nombreux domaines de l'informatique :

- informatique théorique : automates, grammaires, langages
- structures de données : tables de symboles, arbres de syntaxe abstraite
- algorithmes : algorithmes d'optimisation, programmation dynamique, coloration de graphes,
- architecture des ordinateurs : organisation de la mémoire, allocations, sélection d'instructions
- génie logiciel : modularité, utilisation de design patterns, d'un environnement de développement (eclipse)
- logique : systèmes de typage
- intelligence artificielle : heuristiques d'optimisations

D'autres logiciels sont de proches parents des compilateurs : interpréteurs (python, shell), traducteurs de langages naturels, correcteurs d'orthographe. Les techniques utilisées en compilation sont aussi utilisées par les préprocesseurs pour l'expansion des macros, les éditeurs de textes comme emacs, les interprètes de requêtes dans les bases de données. En conclusion, l'étude des compilateurs permet de mieux comprendre un certains nombre d'outils familiers du programmeur, ainsi qu'un certain nombres de notions cruciales telles que la portée des variables, le typage, le polymorphisme.

1.2 Les phases de la compilation

Un compilateur se découpe en **trois grandes parties** :

- le **front-end** : dépendant du langage source

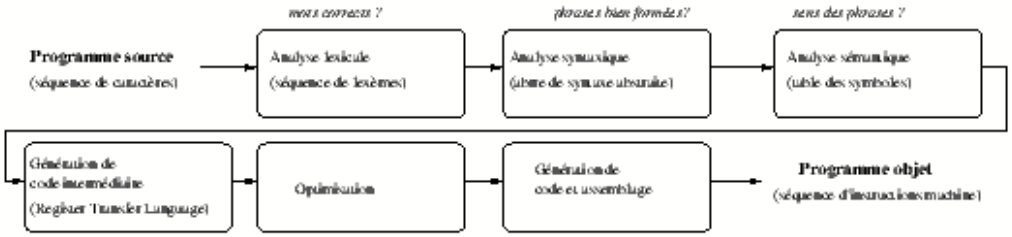


Figure 1: Les étapes d'une compilation.

- le **coeur** qui travaille sur une représentation intermédiaire du programme, indépendante à la fois du langage source et du langage cible
- le **back-end** : fortement lié au langage cible

L'avantage de ce découpage est la réutilisabilité du coeur pour différents langages sources et pour différents langages cibles.

1.2.1 Les différents modules

La production d'un exécutable comprend une succession d'étapes importantes, implémentée par différents modules :

- préprocesseur
- analyseur lexical
- analyseur syntaxique
- gestion d'une tables de symboles
- analyse sémantique, typage
- générateur de code intermédiaire
- optimisation de code
- générateur de code assembleur
- éditeur de liens

Nous distinguons le compilateur des outils qui sont utilisés en amont : Editeur, Préprocesseur, et des outils qui sont utilisés en aval : Assembleur, lieur, chargeur.
Sources -> Préprocesseur -> Programme source -> Compilateur -> programme cible -> Assembleur -> Lieur-chargeur

1.2.2 Les performances

Les performances diffèrent d'un compilateur à un autre :

- Detections des erreurs plus ou moins performante,

– Optimisations de code plus ou moins poussées,
– Sécurité plus ou moins poussée,
– Production d'instructions plus ou moins structurées. L'assembleur par exemple est trés peu structuré, il présente des instructions machines, des directives, des étiquettes.

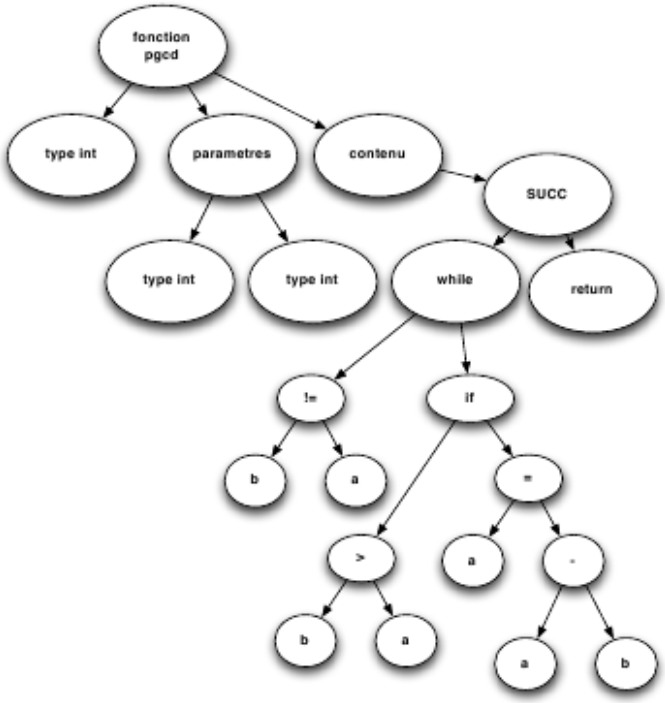
1.2.3 Les Entrées/sorties

En entrée :
– des données : variables, structures, tableaux, pointeurs, classes
– des instructions de haut niveau : boucles, conditionnelles, fonctions
En sortie :
– registres, adresses mémoires, étiquettes
– instructions de bas niveau : lecture/écriture de registres, opérations arithmétiques ou logiques de base, sauts, directives (appel de fonctions)

1.3 Exemple : PGCD

```
int PGCD(int a , int b )
{
while ( b != a ) {
   if ( a > b )
      a = a-b;
   else {
      /* Echanger a et b */
      int tmp;
      tmp = a;
      a = b;
      b = tmp;
      }
   return a;
}
```

1. Analyse lexicale :
   Elle spécifie les règles d'écriture des mots du langage source.
   Identifie les commentaires, mots réservés, opérateurs, constantes, identificateurs.
2. Analyse syntaxique :
   Elle spécifie la manière de construire des textes dans le langage source.



ELle identifie les instructions structurées, les blocs imbriqués, les déclarations, et transforme le code en arbre.

3. Analyse sémantique, génération de code intermédiaire :
   Elle analyse la signification du texte, que représentent les identificateurs, quelles opérations sont associées aux instructions.
   - Types :
   $PGCD\ int \times int \rightarrow int$
   $a\ int$
   $b\ int$
   - Instructions :

```
SUCC(WHILE(TEST (!=, a, b), IF (TEST (>, a, b), AFF (a, OP (-, a, b)),
  SUCC(BLOC(VAR(tmp, int), AFF (tmp, a), SUCC(AFF (a, b), AFF (b,
  tmp))))))),
RETURN (a))
```

Chapitre 2

Analyse lexicale

Rôle : grouper les lettres pour former des mots.
Les mots sont appelés lexèmes (ou token).
Au passage, cette phase :

- peut reconnaître les mots réservés, constantes, identificateurs
- signale les mots mal orthographiés
- peut supprimer les commentaires.
- peut préprocesser le texte : expansion de macros.

2.1 Entités lexicales (token)

Définition par des expressions rationnelles. L'analyseur est un automate fini dont les états terminaux sont associés à des actions (traduction, émission d'un code caractérisant le léxème).

2.1.1 Expression rationnelle

Soit un alphabet A, un ensemble rationnel est :

1. Ensemble fini de mots
2. Concaténation d'ensembles rationnels $R1$ $R2$
3. Itération d'ensembles rationnels R
4. Union d'ensembles rationnels $R1 \cup R2$

Exemple :

```
Alphabet = [A-Z][a-z][0-9]"_";
Lettre = [A-Z][a-z]
Chiffre = [0-9]
```

```
Identficateur = {Lettre} ({Lettre} | {Chiffre} |  "_") *
Entier = {0} | [1-9]{Chiffre}*
```

2.1.2 Automate

Un analyseur lexical s'implémente à l'aide d'un automate fini. L'automate doit être déterministe. Il doit reconnaître l'ensemble des lexèmes. L'automate a un comportement particulier : il ne s'arrête pas au premier état final rencontré, il le mémorise (avec la position dans le texte d'entrée atteinte dans cet état) et continue la reconnaissance jusqu'à l'état final premettant de lire le plus grand nombre de caractères du programme source.

2.2 JFLEX : fonctionnement

Fichier flex -> JFlex -> Lexer.java -> Java -> Lexer.class

L'analyseur lexical reconnait des ensembles de mots. Chaque ensemble de mots (lexème) est décrit par une expression rationnelle ou expression régulière.
L'analyseur lexical associe à chaque expression une action (traduction, spécification du lexème,...). Pour Jflex cette action est du code Java.
L'analyseur lexical produit par Jflex lit le fichier d'entrée, identifie les mots appartenant aux ensembles décrits par les expressions régulières et pour chaque mot reconnu, effectue une séries d'actions écrites en Java dans le fichier source de l'analyseur : `lexer.jflex`.

2.2.1 Format d'un fichier source jflex :

Il contient trois sections séparées par le signe `%%`

- section 1 : le nom du paquetage, les importations d'autres paquetages.
- section 2 : des options (`%cup`, `%public`, `%line`), des définitions de variables et fonctions utiles encadrées par les symboles `%{` et `%}`.
- section 3 : les expressions rationnelles et les actions associées à chaque expression.

```
%{
       // Code de définition java
%}
```

```
<partie définitions>

%%

<partie règles>
       expression { <code java> }

%%

<partie code utilisateur (java)>
```

Dans la classe java Lexer générée par Jflex, la fonction d'analyse lexicale s'appelle yylex().

Exemple d'appel de la fonction d'analyse lexicale :

```
public static void main(String[] args) {
   try {
      FileReader  myFile = new FileReader(args[0]);
      Lexer myLex = new Lexer(myFile);
      return lexer.yylex();
       }
   catch (Exception e){
      System.out.println("invalid file");
        }

}
```

Le texte compris entre `%{` et `%}` est recopié par jflex dans le source java généré avant yylex().

Lorsque yylex() est appelée, elle cherche dans l'entrée la règle dont l'expression reconnaît le plus long motif (en considérant aussi l'état courant). Si plusieurs règles reconnaissent ce plus long motif, la première apparaissant dans le source jflex est appliquée. Si aucune règle ne s'applique, les caractères lus en entrée et qui ne sont pas reconnus sont recopiés sur la sortie.

Lorsque une chaîne de caractères est reconnue par l'analyseur elle est retournée par la fonction $yytext()$, et sa longueur est retournée par la fonction $yylength()$.

2.2.2 Expressions rationnelles dans Jflex :

a : le caractère 'a'

si e,e' sont deux expressions :

$e|e'$: soit e, soit e'

ee' : e suivie de e'

si e est une expression :

$e*$: un nombre arbitraire de e.

$e+$: au moins une occurrence de e.

$e?$: zéro ou une occurrence de e.

$\tilde{e}$: jusqu'à une occurrence de e.

$e\{n\}$: n occurrences de e.

$e\{n,m\}$: de n à m occurrences de e.

2.2.3 Définition dans Jflex :

Les ensembles rationnels peuvent être spécifiés sous la forme de macros :
<nom> <expression>
Dans la partie règles, on pourra utiliser $\{nom\}$ pour désigné l'ensemble rationnel défini par $< expression >$.

Exemple

```
import java.io.*;
%%
%public
%class Lexer
%standalone
%8bit
```

```
%{
   StringBuffer str = new StringBuffer();
%}
LineTerminator = \r|\ n|\ r\ n
InputCharacter = [\^\ n\ r]
WhiteSpace = \{LineTerminator\}|[ \ f\ t]

\%\%
/* Keywords */
if { System.out.printf("KEYWORD:\%s\ n", yytext());}
else {System.out.printf("KEYWORD:\%s\ n", yytext());}

/* Operators */
"+" {System.out.printf("OPERATOR:\%s\ n", yytext());}

/* Literals */
/* Comments and whitespace */
{WhiteSpace} {/* Nothing */}
```

2.2.4 Etats :

Rôle : appliquer des règles de façon conditionnelle.
Si l'analyseur lexical possède des états $E1$, $E2$,..., à tout moment, il se trouve dans un état Ei.
Au départ, il est dans l'état nommé $YYINITIAL$. L'appel de la fonction $yybegin(Ei)$ le met dans l'état Ei. On peut écrire des règles qui ne s'appliquent que dans un état bien précis.

Définition des états (dans la partie 1)
$\%state < nom\_etat >$: état inclusif.
$\%xstate < nom\_etat >$: état exclusif.

Source lex :

```
%{
       // Code java
%}
%state etat1
%xstate etat2
 ...
```

```
%%
  ...
```

Lorsque l'analyseur est dans l'état etat1, les seules règles actives sont :
. celles préfixées par <etat1>,
. celles non préfixées.
Lorsque l'analyseur est dans l'état etat2, les seules règles actives sont celles préfixées par <etat2>.

Exemple :

```
%x SPECIAL
specialmotif  expression1
endmotif      expression2
%%
{specialmotif}   {yybegin(SPECIAL);}
<SPECIAL>blabla  do_something();
{endmotif}       {yybegin(YYINITIAL);}
```

Chapitre 3

Analyse syntaxique

3.1 Introduction

Les langages réguliers sont insuffisants pour spécifier la structure d'un programme. Il est nécessaire de reconnaître des constructions imbriquées, comme les blocs, les boucles, les expressions arithmétiques parenthésées... Pour ces constructions il est nécessaire d'avoir une pile.

Equivalence : langages rationnels $\leftrightarrow$ automates finis
langages algébriques $\leftrightarrow$ automates à pile

Exemple : $\{a^n b^n, n \geq 0\}$ est algébrique, mais n'est pas régulier (a représent par exemple l'accolade ouvrante et b l'accolade fermante).
On peut donner un automate à pile pour reconnaître ce langage : à la lecture de l'accolade ouvrante, il la met sur la pile ; à la lecture de l'accolade fermante, il dépile l'accolade ouvrante correspondante.

On donne au constructeur d'analyseur syntaxique une grammaire algébrique, c'est à dire une liste de règles de grammaire.
Exemple :

```
axiom -> prog <EOF>
       | <EOF>

prog -> decl_list fonc_list
      | fonc_list

decl_list -> decl_list decl
           | decl
```

```
fonc_list  ->  fonc_list fonc
            | fonc

fonc  -> return_type fonc_name '(' args_list ')' bloc

return_type -> int | double | ...
```

On peut représenter les règles utilisées pour construire un programme donné par un arbre dont la racine est `axiom`. Cet arbre est appelé arbre de dérivation ou arbre syntaxique.

Le constructeur d'analyseur syntaxique traduit la grammaire en un automate à pile.
Idée : :
- commencer par mettre le symbole de début `start` sur la pile.
- chaque fois qu'un symbole X est en sommet de pile et qu'il existe une règle `X -> u` dans la grammaire, remplacer X par u sur la pile.
- dépiler chaque fois que le symbole sur la pile est le même que le symbole lu dans le programme source.

Mais l'application naïve de ce principe donne un automate à pile fortement non-déterministe. Il faut donc utiliser des méthodes un peu plus élaborées.

3.2 Trois grandes classes d'analyseurs syntaxique

3.2.1 Analyseurs descendants

L'analyseur essaie de reconstituer l'arbre syntaxique à partir de la racine `start` et de la branche gauche.
→ méthode LL (Leftmost generation of a Leftmost derivation)
Cette méthode ne s'applique qu'avec une classe restreinte de grammaires (ce sont des grammaires non-ambigües et non récursives gauches). Donc ne convient pas pour la plupart des langages de programmation.
L'algorithme de reconnaissance d'un programme est alors en temps linéaire.

3.2.2 Analyseurs ascendants

L'analyseur essaie de reconstituer l'arbre syntaxique à partir des feuilles et de la branche gauche.
→ méthode LR (Leftmost generation of a Rightmost derivation). Ces analyseurs ne s'appliquent également qu'à des grammaires non-ambigües (pas

C++). Mais les logiciels (`cup` ou `yacc`) peuvent traiter des grammaires ambigües si on leur ajoute des règles de priorités.
L'algorithme de reconnaissance d'un programme est alors en temps n^3.

3.2.3 Analyseurs tabulaires

- Coke-Younger-Kasami (bottom-up) : en n^3, peu utilisé en pratique (introduit un certain nombre d'items inutiles).
- Earley (top-down) : en n^3, en n^2 si la grammaire est non-ambigüe, linéaire avec la plupart des grammaires LR. Autre avantage : détecte immédiatement à la lecture d'un symbole, si le mot lu jusqu'alors est prefixe d'au moins un mot engendré par la grammaire.
- GLR (LR Generalisée) utilisée pour les langages naturels (et C++) : en n^3, linéaire si la grammaire est non-ambigüe, (sinon duplication de l'automate si on arrive dans un état non-déterminisme, duplication de la pile aussi mais avec partage de parties communes -> utilise des dags).

3.3 Rôle de l'analyseur

- analyser la structure du programme source.
- relevé les erreurs syntaxiques.
- La récupération sur erreurs : il faut être capable de continuer l'analyse même si le code source contient des erreurs (ou pour les langues naturelles, si la grammaire est incomplète). Il s'agira de détecter les problèmes et produire une analyse même imparfaite. Dans le cas des grammaires ambigües, on peut choisir une tansition de l'automate et continuer l'analyse : si l'automate tombe dans l'état erreur, il faudra revenir en arrière et essayer avec une autre action.

Quatre approches possibles :

- mode panique : supprime 1 à 1 les symboles du progamme source qui font problème jusqu'à se rattraper.
- production d'erreurs (`cup`, `yacc`) : rajoute une règle de grammaire contenant la notion d'erreur, par exemple : `expression -> error ;`. L'analyseur avance alors jusqu'au prochain symbole correct après l'erreur (ici le `;`).
- correction locale (`cup2`) : remplace le symbole lu par un autre qui semble plus juste.
- correction globale : trouve le plus petit ensemble d'insertions ou délétions de symboles tel que l'analyse soit correcte (même si ces insertions

ou délétions ne sont pas à l'endroit même de l'erreur). Exemple : algorithme de Burke-Fisher -> pas d'insertions ou délétions au delà de k tokens avant l'erreur, avec k fixé.

3.4 Grammaires algébriques

Moyen de décrire quels sont les flux de lexèmes corrects et comment ils doivent être structurés. l'analyseur lexical décrit comment former les mots du langage. L'analyseur syntaxique décrit ensuite comment assembler les mots pour former des phrases correctes.
La grammaire fournit une description des phrases (= programmes) syntaxiquement corrects.

3.4.1 Definition

$G = \{T, V, S, R\}$
T : alphabet des terminaux
V : alphabet des variables
$S \in V$: symbole de départ ou axiome de la grammaire
R : ensemble fini de règles. Chaque règle $r \in R$ est de la forme : $X \to u$, où $X \in V$ et $u \in (T \cup V)^*$.

Une grammaire sert à décrire un langage. Pour engendrer un langage, on part d'un symbole particulier de la grammaire appelé axiome, généralement noté S.
On appelle dérivation de u à partir de A et on note $A \to^* u$, si u est obtenu à partir de A par l'application séquentielle d'un nombre fini de règles de G.
Langage engendré par la grammaire G : $L(G) = \{u \in T^* | S \to^* u\}$

L'analyseur, connaissant la grammaire G :
- reçoit en entrée un mot u sur l'alphabet terminal, qui correspond au programme source.
- reconstitue une dérivation de u, si le mot u est engendré par G.
- indique une erreur syntaxique sinon.

Exemple : la grammaire "ETF" pour engendrer des expressions arithmétiques comme a + b * (c + d) :
E -> E + T
E -> T

T -> T * F
T -> F
F -> (E)
F -> IDENTIFIER

3.4.2 L'arbre de dérivation

Il traduit l'application des règles dans une dérivation de u à partit de l'axiome S. La racine est l'axiome, les noeuds interne sont les variables (ou non-terminaux) de la grammaire, les feuilles sont les terminaux (correspondants aux lexèmes fournis par l'analyseur lexical).
Remarque : l'arbre de dérivation ne reflète pas l'ordre d'application des règles : un même arbre de dérivation traduit une dérivation gauche (où l'on applique toujours les règles au non-terminal le plus à gauche) et une dérivation droite (où l'on applique toujours les règles au non-terminal le plus à droite).
L'analyseur syntaxique tente de reconstituer un arbre de dérivation pour le mot (le programme) lu en entrée. S'il commence par la racine de l'arbre, on parle d'analyse top-down, s'il commence par les feuilles de l'arbre on parle d'analyse bottom-up.

Exemple : (méthode ascendante)
a + b * (c + d)
F + b * (c + d)
T + b * (c + d)
E + b * (c + d)
E + F * (c + d)
E + T * (c + d)
E + T * (F + d)
E + T * (T + d)
E + T * (E + d)
E + T * (E + F)
E + T * (E + T)
E + T * (E)
E + T * F
E + T
E

3.4.3 Ambiguïté

Une grammaire est ambigüe si il existe un mot u qui possède deux arbres de dérivation différents dans G, c'est-à-dire s'il existe deux structures grammaticales possibles donnant le même mot.

Exemple : expressions arithmétiques :
E -> E + E
E -> E * E
E -> IDENTIFIER

L'analyse pose problème si la grammaire est ambiguïe. On recherche alors une autre grammaire engendrant le même langage et qui soit non-ambigüe.

Exemple de la grammaire "ifthenelse" :
instr -> si expr alors instr
instr -> si expr alors instr sinon instr

Deux analyses possibles de la phrase :
"si E_1 alors S_1 sinon si E_2 alors S_2 sinon S_3"
On préfère le "alors" le plus proche (p.201)

Grammaire équivalente :
instr -> instr_close
instr -> instr_non_close
instr_close -> si expr alors instr_close sinon instr_close
instr_non_close -> si expr alors instr_non_close
instr_non_close -> si expr alors instr_close sinon instr_non_close

3.5 Notions de grammaire attribuée

Le rôle de l'analyse syntaxique n'est pas seulement d'identifier les phrases correctes dans la grammaire du langage. C'est aussi de collecter un certain nombre d'informations utiles pour la phase suivante de la compilation : l'analyse s'émantique. Ces informations sont attachées aux symboles de la grammaire.

Chaque noeud de l'arbre syntaxique peut porter de l'information appelée **attribut**. Pour les feuilles de l'arbre, l'analyseur lexical peut fournir un at-

tribut associé à un lexème donné (par exemple la valeur pour une constante entière).
Pour calculer les attributs des noeuds interne de l'arbre, l'analyseur syntaxique associe à chaque règle de grammaire une **action sémantique**. Cette action indique comment calculer l'information attachée à un symbol de la grammaire (ie : un noeud de l'arbre syntaxique) en fonction de l'information attachée aux autres symboles présents dans la règle.
On appelle **traduction dirigée par la syntaxe** le fait d'attacher de actions sémantiques aux règles de la grammaire.
Une **grammaire attribuée** est une grammaire dans laquelle :

- chaque symbole, terminal ou non, peut avoir des attributs,
- chaque attribut possède des règles de calcul en fonction d'autres attributs et de valeurs initiales,
- chaque règle de calcul est attachée à une règle de la grammaire.

But : calculer les attributs pendant l'analyse syntaxique.

Principe :

L → E	imprimer(E.val)
E → E + T	E.val = E1.val + T.val
E → T	E.val = T.val
T → T * F	E.val = E1.val * T.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → IDENTIFIER	F.val = IDENTIFIER.vallex

Soit la règle calculant l'attribut a associée à la production $A \rightarrow u$:
$a = f(a_1, ..., a_n)$
a est **synthétisé** si c'est un attribut de A calculé en fonction des attributs des symboles de u. Une grammaire est **S-attribuée** si tous ses attributs sont synthétisés. L'information remonte dans l'arbre syntaxique.
Pour une grammaire S-attribuée, le calcul des attributs se fait de façon mécanique si on a un analyseur LR (ex :CUP). Lors de la réduction par $A \rightarrow u$ on calcule le (ou les) attribut(s) de A en fonction des attributs des symboles de u. On peut empiler les attributs calculés au cours de l'analyse. Les attributs des tokens sont initialisés par l'analyseur lexical.

Soit la règle calculant l'attribut a associée à à la production $A \rightarrow u$:
$b = f(a_1, ..., a_n)$
L'attribut b est **hérité** si b est un attribut d'un symbole X_j de u, calculé en fonction des attributs d'autres symboles de u ou du symbole A.

Une grammaire est **L-attribuée** si l'attribut b du $j$ème symbole X_j est calculé en utilisant seulement les attributs hérités de $X_1, ..., X_{j-1}$ et celui de A. L'information descend dans l'arbre syntaxique.

L'utilisation des attributs permet une traduction dirigée par la synatxe. Cette technique est utilisée par le compilateur pour effectuer le contrôle des types, l'évaluation des expressions, la construction de représentations intermédiaires.

3.6 Le logiciel CUP

Logiciel pour construire automatiquement un analyseur syntaxique.
Format d'un source cup :

```
// imports de fichiers et paquetage

/* The grammar terminals */
terminal  type_1 token_name1, token_name2, ...;

/* The grammar non terminals */
non terminal type_2 name1, name2, ...;

/* The grammar rules */
start with name1;

name1 ::=  ...  {:   /*code Java */   :}

...
```

Le nom des terminaux de la grammaire doit être donné dans le fichier CUP, par exemple `ID` pour un identificateur. Il doit être le même que celui utilisé par le lexer lors de l'appel du constructeur de Symbol, par exemple `CalculetteSymbol.ID`. Le logiciel CUP génère automatiquement un fichier `CalculetteSymbol.java` qui associe un code (i.e. une constante entière) à chaque terminal de la grammaire :
- vérifier qu'il y a autant de constantes définies dans ce fichier que de terminaux déclarés dans cup par le mot-clé `terminal`,
- vérifier que pour chacun de ces terminaux, le lexer effectue un appel au constructeur de Symbol (dans le fichier jflex).

L'analyse syntaxique est donc nécessairement précédée d'une analyse lexicale. Le constructeur du parseur prend donc en paramètre un analyseur

lexical (de la classe Lexer). La méthode appelée pour effectuer l'analyse syntaxique s'appelle *parse*().

Exemple de construction d'un parseur :

```
public static void main(String[] args) {
  FileReader  myFile = new FileReader(args[0]);
  Lexer myLex = new Lexer(myFile);
  Parser myP = new Parser(myLex);
  try {
      result=myP.parse();
   }
  catch (Exception e) {
      System.out.println("parse outor...");
   }
}
```

Chaque fois que la méthode *parse*() a besoin d'un nouveau terminal, elle appelle la méthode de l'analyseur lexical *yylex*(). Lorsque yylex reconnaît un lexème, elle retourne au parseur une instance de la classe Symbol pour décrire le type de lexème rencontré.
Le symbole transmis peut aussi contenir des informations (du type Object) sur le lexème reconnu, par exemple son nom ou sa valeur. Ces informations constitue ce que l'on appelle l'**attribut** du symbole.

3.6.1 Les attributs dans jflex et CUP

Dans jflex, les lexèmes ou tokens sont représentés par des instances de la classe Symbol. Les attributs d'un token sont stockés dans une variable d'instance de cette classe. Le constructeur de Symbol, qui est appelé lors de la reconnaissance du token dans le flux d'entrée, peut prendre en paramètre une valeur (appartenant à une sous-classe de la classe Object) :

```
Symbol symbol (int type, Object value);
```

Cette valeur permet l'initialisation de l'attribut du token. Le token (instance de Symbol) construit est alors transmis (par un "return") à la fonction d'analyse syntaxique.

Les non terminaux de la grammaire peuvent également avoir des attributs. Lorsqu'un non terminal figure à gauche d'une règle, son attribut est

automatiquement désigné par la variable $RESULT$. Cette variable n'a pas besoin d'être déclarée, mais il faudra l'initialiser dans le code java associé à la règle.
Soit la règle : $F \rightarrow ID$
. Pour le symbole F à gauche, son attribut est noté $RESULT$
. Pour associer un attribut au symbole ID à droite, il suffit de le nommer (identificateur de votre choix) :
$F \rightarrow ID : val$
- l'attribut de ID est alors val.

Soit la règle : $E \rightarrow E + T$
. Pour le symbole E à gauche, son attribut est noté $RESULT$
. Pour associer un attribut aux symboles à droite, il suffit de les nommer :
$E \rightarrow\ E : eval\ +\ T : tval$
- l'attribut du E à droite est $eval$
- l'attribut de T est $tval$
Si les attributs sont des nombres, on peut alors initialiser RESULT avec la valeur $eval + tval$.
Il est également possible de savoir quel portion de l'entrée est reconnue par le non terminal T, par exemple. En effet deux autres variables sont associées à T en plus de son attribut $tval$, elles sont nommées automatiquement $tvalleft$ et $tvalright$ (de type int) et indiquent deux positions dans le texte en entrée.
On peut choisir un type pour l'attribut associé à chaque symbole de la grammaire :

```
terminal     type_1   token_name1;
non terminal type_2   name2;
```

Ces types doivent être des classes Java.

Exemple :
- si le type associé à `expr` est `Integer` :

```
expr ::= expr:eval PLUS term:tval
            {: RESULT = new Integer(eval + tval); :}
```

- si le type associé à `ID` est `String` :

```
factor ::= ID:t
           {: System.out.println(" Attribut de ID: " + t ; :}
```

3.6.2 Priorités des opérateurs

Définir des priorités et associativités pour les lexèmes permet d'obtenir un comportement autre que celui choisi par défaut par cup en cas d'ambigüité. La priorité d'un lexème est déterminée par la position de sa définition : plus il est défini tard, plus forte est sa priorité.

Un lexème est :
- associatif à gauche s'il a été défini par
  `precedence left OPERATEUR;`
- associatif à droite s'il a été défini par
  `precedence right OPERATEUR;`

On peut mettre plusieurs token dans une même déclaration de precedence, ils ont alors même priorité.
La priorité d'une règle est celle de son terminal le plus à droite. Si la règle n'a aucun terminal, sa priorité est plus basse. Il est aussi possible de modifier la priorité d'une règle, par une directive `%prec OPERATEUR` placée après la règle et ses actions. La règle aura alors le même priorité que `OPERATEUR$`

Exemple :
```
precedence right EGAL;
precedence left PLUS, MOINS;
```
On peut déduire de ces définitions que :
- les token `PLUS` et `MOINS` ont la même priorité,
- le token `EGAL` a une priorité plus faible que les token `PLUS` et `MOINS`

3.7 Principe de l'analyse LR

Principe : construire l'arbre de bas en haut, en partant de la chaîne de terminaux.
Objectifs :
- deviner les règles de grammaire conduisant à une analyse correcte de la chaîne de terminaux,
- obtenir un algorithme déterministe,
- pas de backtraking.

Idée : l'analyseur mémorise (à l'aide d'une pile) la partie de l'arbre de dérivation déjà construite, et aussi ce qu'on doit arriver à lire dans le futur pour parvenir à une analyse correcte.
Il peut ensuite comparer avec les symboles suivants sur le flux d'entrée :

lors d'une analyse $LR(k)$, il compare avec les k symboles suivants sur le flux d'entrée.

Pour mémoriser ce qui a été analysé et quelles sont les suites possibles de l'analyse déjà effectuée, on utilise des règles pointées (c'est-à-dire des règles de grammaire dans lesquelles on rajoute un point en partie droite) appelées **items**. Le point indique quelle portion de la règle a déjà été utilisée pour reconnaître l'entrée lue jusqu'à présent.
Si une règle peut s'écrire $X \rightarrow uv$ où u et v sont deux mots, alors $X \rightarrow u.v$ est un item, avec le point entre u et v.
L'analyseur alterne des étapes d'**empilement** (en anglais **shift**), qui consistent à transférer sur la pile le terminal lu sur le flux d'entrée, et des étapes de **réduction** (en anglais **reduce**), qui consistent à remplacer les items en sommets de pile par de nouveaux items résultant de l'application d'une règle de grammaire (sans consommer le flux en entrée). Les étapes de réduction permettent de "remonter" dans l'arbre de dérivation. A chaque réduction, on empile un nouvel ensemble d'items sur la pile. Ces nouveaux items traduisent toutes les analyses futures possibles.

Exemple :

1. E -> • E + T
   E -> • T
   T -> • T * F
   T -> • F
   F -> • (E)
   F -> • IDENTIFIER
2. F -> IDENTIFIER •
   T -> F •
   E -> T •

3. F -> (• E)
   E -> • E + T
   E -> • T
   T -> • T * F
   T -> • F
   F -> • (E)
   F -> • IDENTIFIER

4. F -> (E •)
   E -> E • + T
   ...

3.7.1 Application dans Cup

Le logiciel CUP doit construit un automate déterministe. Dans chaque état, le symbole lu sur le flux d'entrée doit déterminer quel action (**shift** ou **reduce**) effecter et dans quel nouvel état se placer. Si la grammaire est ambigüe, CUP ne pourra pas construire l'automate déterministe.
Il affichera alors :
- soit un conflit **shift**/**reduce** indiquant que deux actions différentes sont possibles pour la lecture d'un même symbole en entrée (consommation de l'entrée ou application d'une règle de grammaire),
- ou un conflit **reduce**/**reduce** indiquant que deux règles de grammaire s'appliquent, ce qui va correspondre à deux futurs différents possibles dans la suite de l'analyse.

Il faut alors modifier les règles de grammaire pour supprimer toute ambiguïté.

Pour avoir le détails de l'automate construuit, on peut utiliser les options suivantes de CUP :
-dump_grammar
-dump_states
-dump_tables
-dump
Ces options produisent un affichage de la grammaire, des états de l'automate (utile pour repérer les conflits), des tables de transitions de l'automate. La dernière option est identique à l'utilisation simiultanée des trois précédentes.

3.8 Algorithme d'Earley

... A compléter ...

3.9 Le traitement des erreurs de syntaxe

L'approche choisie dans les analyseurs que nous construirons est la possibilité d'utiliser une production d'erreurs.

Lorsque l'analyseur rencontre une erreur :

1. en l'absence du symbole `error` dans les règles de grammaire, il arrête l'analyse.
   Par contre si le symbole `error` figure dans les règles :
2. il dépile les symboles analysés jusqu'à ce qu'il puisse empiler `error`, et se place dans un mode spécial appelé mode "reprise d'erreur",
3. il consomme en entrée les symboles qui ont provoqué l'erreur (comme si l'entrée lue était le symbole `error`),
4. si dans ce mode spécial, il rencontre une autre erreur avant d'avoir pu empiler un certain nombre de symboles (par défaut 3 dans CUP) il revient en (2).

Attention : l'introduction du symbole `error` dans la grammaire peut créer des conflits.
Si un nombre suffisant de tokens ont pu être reconnus après le texte correspondant à `error`, l'analyseur sort du mode "reprise d'erreur". Dans CUP ce nombre suffisant de token est fixé par la méthode `error_sync_size()` (qui retourne 3 par défaut). Le symbole `error` est un terminal spécial. Il n'a pas besoin d'être déclaré dans CUP.

Exemple :

```
statement :== expr SEMICOL | while_stmt SEMICOL | if_stmt SEMICOL | ... |
            | error SEMICOL
               {: System.err.println("syntax error before ';'"); :}
```

Cette règle indique à l'analyseur, que si aucune des règles normales de `statement` ne s'appliquent, une erreur de syntaxe doit être signalée et il doit lire l'entrée jusqu'au prochain ';' puis continuer comme si une instruction venait d'être reconnue, ce qui correspond à une réduction par la règle `instruction -> error SEMICOL`.
On pourra connaître le texte correspondant au token spécial `error` en lui ajoutant un attribut `e` et en utilisant les deux variables vues précédemment `eleft` et `eright`.

Du point de vue de l'analyse LR, si le parser rencontre un token qui ne lui permet pas d'avancer sur un nouvel état de l'automate, il revient en arrière jusqu'au dernier état rencontré contenant un item de la forme `X -> u.error v` . Le parser passe alors dans l'état contenant l'item `X -> u error .v` comme s'il avait lu le token "error" en entrée et il consomme tous les symboles sur le flux d'entrée jusqu'à ce qu'il puisse lire `v`.

D'autre méthodes du parser peuvent être appelée pour le traitement des erreurs. Elles effectuent par défaut un traitement de base. Pour un traitement plus fin, il sera nécessaire de les redéfinir :

`public void report_error(String message, Object info);`
cette méthode doit être appelée pour signaler une erreur syntaxique. Par défaut le message est affichée sur la sortie d'erreur et le deuxième paramètre est ignoré.

`public void report_fatal_error(String message, Object info);`
cette méthode doit être appelée pour signaler une erreur non récupérable. Elle appele d'abord la méthode `report_error` puis la méthode `done_parsing`, qui arrête le parseur, et enfin elle lève une exception.

`public void syntax_error(Symbol cur_token);`
cette méthode est appelée par le parseur chaque fois qu'une erreur est détectée. Par défaut elle appelle simplement `report_error("Syntax error", null)`

`public void unrecovered_syntax_error(Symbol cur_token);`
cette méthode est appelée par le parseur chaque fois qu'il est incapable de reprendre après une erreur. Par défaut elle appelle simplement
`report_fatal_error("Couldn't repair and continue parse", null)`

Chapitre 4

Arbres de syntaxe abstraite

L'arbre de dérivation construit par l'analyseur syntaxique possède de nombreux noeuds superflus, qui n'apportent pas d'information sur la sémantique du programme. La nécessité d'utiliser une grammaire non-ambigüe conduit souvent à introduire des symboles auxiliaires dans la grammaire qui ne seront pas utiles à l'analyse sémantique.

Pour faciliter la phase d'analyse sémantique il est donc souhaitable d'utiliser une représentation plus synthétique et qui soit indépendante du langage source.

4.1 Les Représentations Intermédiaires

Nécessité de construire une *Représentation Intermédiaire*, IR en abrégé, du code source qui synthétise toutes les informations nécessaires à l'analyse ultérieure et qui soit indépendante du langage source. Nous utiliserons une IR sous forme d'arbres appelés **Arbres de Syntaxe Abstraite**.
Un compilateur utilise en général plusieurs représentations intermédiaires au cours de ses différentes phases, en allant de la plus structurée, par exemple les arbres de syntaxe abstraite, à la plus simple, par exemple des listes d'instructions en pseudo-assembleur.
On utilise la **traduction dirigée par la syntaxe** pour construire la première Représentation Intermédiaire du programme, pendant l'analyse syntaxique du code source.

4.1.1 Différentes représentations du code source

1. **arbre de dérivation :**
   - lié à la grammaire
   - utilisé dans des compilateurs simple passe ou des interprêteurs
   - permet d'analyser les codes syntaxiquement corrects

2. IR de haut niveau : **arbre de syntaxe abstraite** ou AST
   - indépendant de la grammaire ⇒ coeur du compilateur réutilisable
   - utilisé dans compilateurs multi-passes ⇒ plusieurs parcours de l'AST
   - permet de gérer les environnements et de contrôler les types
   - ne permet pas toutes les optimisations de code

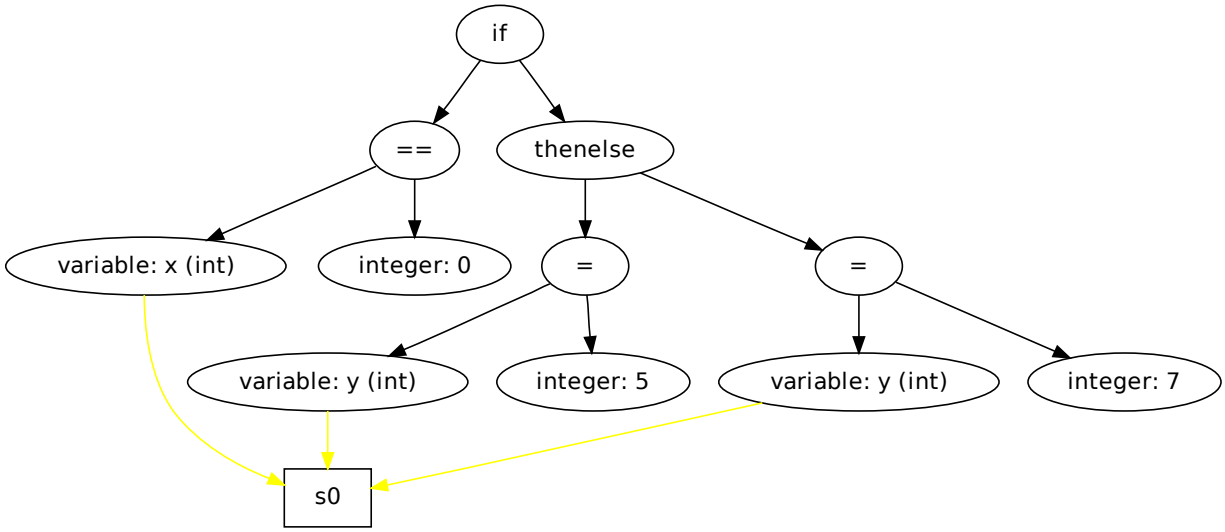
3. IR de bas niveau : **code trois adresses**
   - ne garde plus trace de la structure du programme source
   - code linéaire ⇒ plus d'optimisations (réorganisation du code, suppression d'instructions redondantes)
   - permet le calcul du nombre de registres utiles

4.2 Représentation arborescente

4.2.1 Définition

L'**arbre de syntaxe abstraite** est un arbre qui ne garde plus trace des détails de l'analyse syntaxique, c'est-à-dire qui est indépendant de la grammaire, mais qui mémorise la structure du programme et les actions qui le composent.

Ces arbres permettent de s'affranchir des spécificités du langage source, pour ne garder qu'une représentation générique du code source. Par exemple, dans l'arbre de syntaxe abstraite, les symboles de ponctuation, la notation spécifique des opérateurs algébriques auront disparu. Tout ce qui est dépendant du langage source doit être éliminé de l'arbre de syntaxe abstraite. Un même arbre de syntaxe abstraite doit être associé à un même programme, qu'il soit écrit en C, en python ou en Java.



4.2.2 Construction de l'arbre de syntaxe abstraite

- élimination des symboles de ponctuation, de tabulation,
- élimination de tokens propres au langage source (opérateurs spécifiques, mots-clé spécifiques,...)
- prise en compte de la structure du programme (blocs, boucles,...)
- plusieurs solutions sont possibles

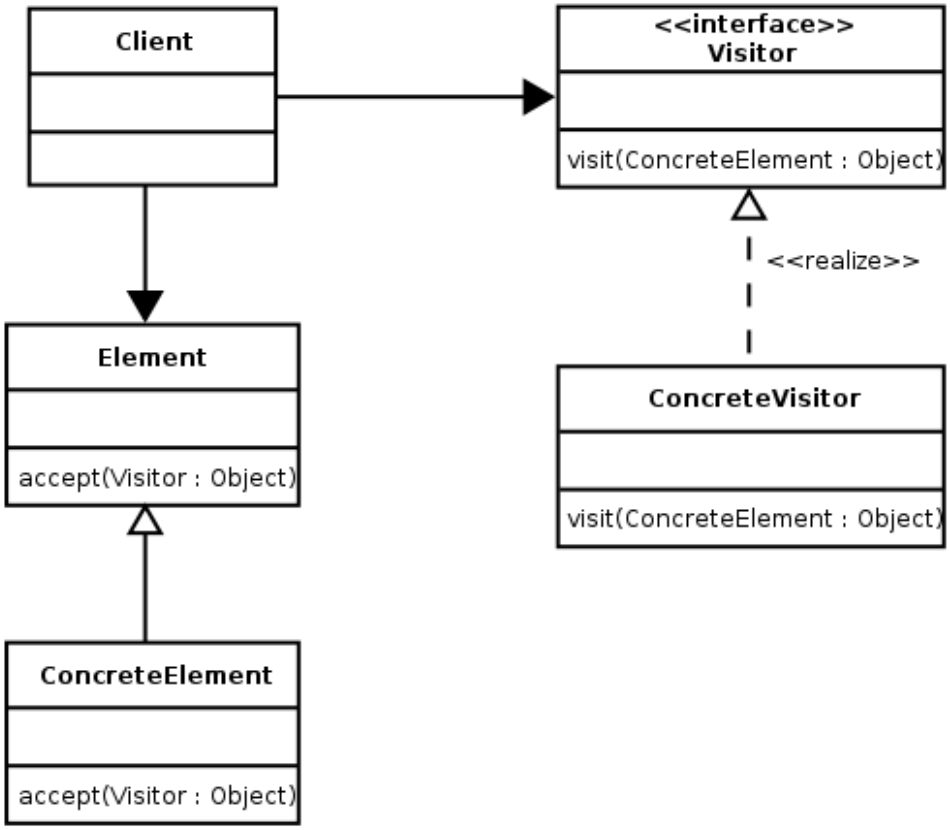
Il existera plusieurs sortes de noeuds dans d'AST : des noeuds "function", des noeuds "variable", des noeuds pour différentes sorte de constantes (entières, réelles, ...) des noeuds pour différentes sorte d'instruction (if, while,...) des noeuds pour différentes sorte d'opérateur (egal, plus,...).

L'arbre de syntaxe abstraite sera construit par l'analyseur syntaxique. Les actions sémantiques associées aux règles de grammaires serviront donc à la construction des noeuds de l'arbre de syntaxe abstraite. Les informations contenues dans un noeud de l'AST seront entre autres des informations de type, de valeur (pour les expressions), des chaînes de caractères (pour les identificateurs ou les opérateurs).

L'analyse sémantique sera effectuée ensuite, par une succession de traitements appliqués aux arbres de syntaxe abstraite. Les traitement appliqueés aux AST seront par exemple l'évaluation, la vérification de type, la traduction dans une autre forme de représentation intermédiaire (code 3 adresses).

4.3 Implémentation

Dans le cas d'un compilateur écrit en java, deux approches sont possibles, l'approche fonctionnelle ou l'approche objet.



4.3.1 Approche fonctionnelle

Dans un compilateur écrit en Java, les noeuds de l'arbre de syntaxe abstraite seront implémentés à l'aide de classes abstraites et de sous-classes :
- `abstract class Statement`, avec une sous-classe pour chaque alternative :
  - `Block`
  - `If`
  - `While`
  - ...
- `abstract class Expr`, avec une sous-classe pour chaque alternative :
  - `Plus`
  - `Times`
  - `Ident`
  - ...

Chaque traitement correspond à une unique fonction qui reçoit en paramètre la classe de noeud à traiter. En java on utilise le design pattern **Visiteur** (interface Visitor). Il existera un visiteur pour évaluer (classe Interpreter), un visiteur pour vérifier les types (classe TypeVerificator, un visiteur pour traduire en code intermédiaire (classe Traductor). Chaque visiteur doit surcharger sa méthode de visite pour qu'elle puisse prendre en paramètre toutes les sous-classes de noeuds qui apparaissent dans l'AST.

Inconvénient : trés grand nombre de sous-classes, donc trés grand nombre de surcharges.

4.3.2 Approche Objet

La classe principale des noeuds de l'arbre de syntaxe abstraite sera la classe `ArbSynt`.
Chaque sommet porte une étiquette appartenant à un ensemble prédéfini d'étiquettes (implémentées par la classe `EnumTag` en Java). On doit avoir une étiquette pour indiquer une boucle, étiquette pour indiquer un identificateur, une étiquette pour chaque opérateur binaire, ...

Variables d'instance :
. etiquette
. fils droit, fils gauche
. type
. nom (String)
. valeur

Méthodes :
. affichage (toString)
. evaluation
. contrôle de type
. traduction en code intermédiaire

On pourra spécialisée la classe `ArbSynt` avec des sous-classes correspondant aux différentes catégories de noeuds : Declaration, Instruction, Expression,...
On pourra alors utiliser l'**approche objet** pour implémenter les traitements : méthodes abstraites dans la super-classe, puis concrètes dans les sous-classes. Chaque sous-classe peut implémenter des traitements qui lui sont propres.

4.3.3 Extensibilité :

Ajout de nouveaux noeuds :
. plus facile avec l'approche objet : il suffit d'ajouter une classe,
. avec l'approche fonctionnelle : il faut modifier chaque fonction de traitement.

Ajout de nouveaux traitements :
. plus facile avec l'approche fonctionnelle : il suffit d'ajouter une nouvelle fonction de traitement,
. avec l'approche objet : il faut rajouter le traitement dans chaque classe.

4.4 Arbre de syntaxe abstraite décoré

L'arbre de syntaxe abstraite sera décoré avec des attributs :
Chaque sommet est décoré d'un certain nombre d'informations (ses attributs)) : type, identificateur, valeur,... Ces informations sont mises à jour au cours de l'analyse syntaxique (traduction dirigèe par la syntaxe) en utilisant les règles de calcul d'attributs.
Ces informations sont ensuite utilisées pour les différents traitements appliqués à l'arbre de syntaxe abstraite . Plusieurs parcours de l'AST (parcours en profondeurs) seront effectués dans les compilateurs **multi-passes** pour les traitements suivants : affichage, évaluation, vérification de types, optimisations, traduction en code intermédiaire (code trois adresses).

Chapitre 5

Analyse sémantique : Tables de Symboles, Environnements

Pour l'évaluation, la vérification d'un programme, il est nécessaire de mémoriser tous les identificateurs qu'il utilise.
Nous prendrons le cas d'un langage de programmation tel que les programmes comportent des parties déclarations et des parties exécutables (C, C++, JAVA). Les identificateurs (de variables, de fonctions, de classes) devront être déclarés. Ils seront alors stockés dans des tables de symboles. Le compilateur utilisera les tables de symboles pour vérifier la cohérence des valeurs et des types dans le programme et signaler les erreurs de sémantique.

5.1 Tables de Symboles

La table de symboles est remplie pendant la compilation des parties contenant des déclarations :

- définition de classes (en Prog. Objet),
- définition de fonctions,
- déclaration de variables.

La table de symboles est consultée pendant la compilation des parties exécutables :

- accès à une instance de classe (en Prog. Objet),
- appel de fonction,
- accès à une variable,
- référence à une variable (pointeurs).

La structure de données utilisées pour représenter la table des symboles doit être performante, car le compilateur passe un temps important à la consulter.

Les Tables de Symboles rassemblent toutes les informations utiles concernant les variables et les fonctions ou classes du programme. Différentes tables (reliées entre elles) pourront être associées aux différentes classes ou modules du programme.
Pour toute **variable**, la table de symboles garde l'information de :

- son nom,
- son type,
- son accès (adresse en mémoire ou *offset* dans le bloc d'activation d'une fonction),
- sa portée.

Pour toute **fonction**, la table de symboles garde l'information de :

- son nom
- le nom et le type de ses arguments, ainsi que leur mode de passage
- le type du résultat qu'elle fournit
- son code
- son niveau d'imbrication (*level*) dans les appels de fonction imbriqués,
- sa portée.

La table des symboles est construite lors du parcours des parties contenant des déclarations dans l'arbre de syntaxe abstraite. Chaque déclaration d'un nouvel identificateur ajoute une entrée à la table de symboles.

La table de symboles est consultée lors du parcours des parties contenant des instructions dans l'arbre de syntaxe abstraite. Elle est utilisée pour signaler toutes les erreurs concernant les déclarations de variables et l'usage des identificateurs :
variable initialisée avant d'être déclarée, variable lue en dehors de sa portée, type attendu non conforme au type réel de la variable, débordement de tableau,...

Exemple

```
int PGCD(int a , int b )
     /* ajout de la fonction PGCD dans le table de symboles,
        avec deux arguments int et type de retour int */
{
   while ( b != a ) {
      if ( a > b )
```

```
         a = a-b;
      else {
         int tmp = a; /* ajout de la variable tmp dans la table de
                         symboles, avec le type int  et la valeur a */
         a = b;
         b = tmp;
       }
      /* suppression de la variable tmp de la table de symbole */
      }
      return a;
}
```

5.2 Déclarations, Portées

Les types des identificateurs et des expressions se déduisent des déclarations (de classes, de variables, de fonctions, ...). Les déclarations sont valides dans leur **portée**.

Définition : On appelle **portée d'un identificateur** la zone du programme durant laquelle il est visible.

Les identificateurs ayant même portée sont rassemblés dans un même ensemble.

Exemple :

```
{ int i=0;  /* début de la première portée,  ajout de la variable i */

  if (tab[i]==0){  /* deuxième portée */
       int i;      /* nouvelle variable dans la deuxième portée */
       i=readint();
       ...
      } /* fin de la deuxième portée */
  ...
  i++; /* incrémentation de la variable i de la première portée */
  ...
} /* fin de la première portée */
```

De même que les blocs du programme peuvent être emboités, les portées aussi peuvent être emboitées. Des déclarations peuvent être masquées par

d'autres déclarations effectuées ultérieurement dans une autre portée (ou même parfois à l'intérieur de la même portée).

La table de symbole doit refléter l'imbrication des portées. Cela signifie donc que l'implémentation des tables de symboles devra faire intervenir des piles. La table de symboles devra permettre de retrouver à chaque pas du programme les variables de la portée courante et celles des portées englobantes.

5.3 Environnements

Les **environnements** sont des ensembles d'association

`identificateur` $\rightarrow$ `descripteur`,

où les descripteurs sont des ensembles d'informations associées aux identificateurs (nom, type, ...).
Les informations stockées dans les environnements sont collectées lors des déclarations des identificateurs. La plus importante parmi ces informations est le type de l'identificateur. Dans le cas où l'identificateur correspond à une variable qui a une valeur, on stokera également la valeur (ou le moyen d'y accéder en mémoire). Dans le cas d'un tableau, on stockera aussi sa taille. Dans le cas d'un fonction, on stockera le nombre et le type de ses arguments.

Les environnements sont implémentés par les **tables de symboles**. Ce sont des structures de données complexes faisant intervenir plusieurs tables qui peuvent se référencer mutuellement. On peut avoir une table pour les identificateurs des types nommés (exemple : `typedef struct cellule*liste`), une autre table pour les identificateurs de fonctions, une troisième table pour les identificateurs de variables de blocs... Pour compiler du code Java, on utilisera une table de symboles pour les variables d'instance d'une classe et une autre table de symboles pour les méthodes de la classe. La table des méthodes pourra contenir des pointeurs vers la table des variables d'instance.

Lors de la **déclaration** d'un nouvel identificateur, on doit l'**ajouter** à l'environnement courant.
Lors de l'**affectation** d'une valeur à un identificateur, on doit le **chercher** dans l'environnement courant :
- s'il n'est pas présent, on doit signaler une erreur,
- s'il est présent, on ajoute la valeur au descripteur de l'identificateur.

Lors de la **référence** à un identificateur, on doit le **chercher** dans l'environnement courant :

- s'il n'est pas présent, on doit signaler une erreur,
- s'il n'a pas été initialisé, on doit signaler une erreur,
- sinon on retourne sa valeur dans l'environnement courant.

5.4 Implémentation d'une table de symbole

Considérons une table de symboles pour la fonction principale du programme. Plusieurs implémentations sont possibles. Le choix de la structure de données est important pour la rapidité du compilateur.

On associe à la fonction principalke du programme source une **pile de portées**. Lorsque l'on sort d'une portée, on doit dépiler tous les symboles qui ne peuvent plus être utilisés. L'opération "dépiler" doit être facile à implémenter. La recherche d'un identificateur (opération fréquemment utilisée pour la compilation des instructions) doit être efficace.

5.4.1 Une pile de tables de hachage :

On empile une nouvelle table de hachage à chaque fois que l'on rentre dans une nouvelle portée.
Avantage : gestion facile des portées.
Inconvénient : la recherche d'un identificateur se fait en parcourant toutes les portées de la pile, elle peut donc être trés lente, s'il n'est pas dans la portée courante. Accès au pire en $O(n)$ pour une table de n identificateurs.

5.4.2 Une table de hachage contenant des piles :

Chaque clé de la table correspond à un identificateur auquel est associé une pile de descripteurs. Les informations sur l'identificateur s'empilent, de la portée la plus externe à la portée la plus interne. Le sommet de pile correspond à la définition la plus récente de l'identificateur. Lorsque l'on sort d'un bloc, on doit effacer tous les identificateurs qui ne sont plus valables. Nécéssité de repérer les identificateurs appartenant à une même portée, par exemple en ajoutant dans leurs descripteurs un numéro de portée.
Avantage : recherche rapide (en temps constant) d'un identificateur.
Inconvénient : nécessité de retrouver les identificateurs d'une même portée lorsque l'on sort d'un bloc. Il faut parcourir toutes les entrées de la table, donc la suppression peut être lente. Suppression au pire en $O(n)$ pour une

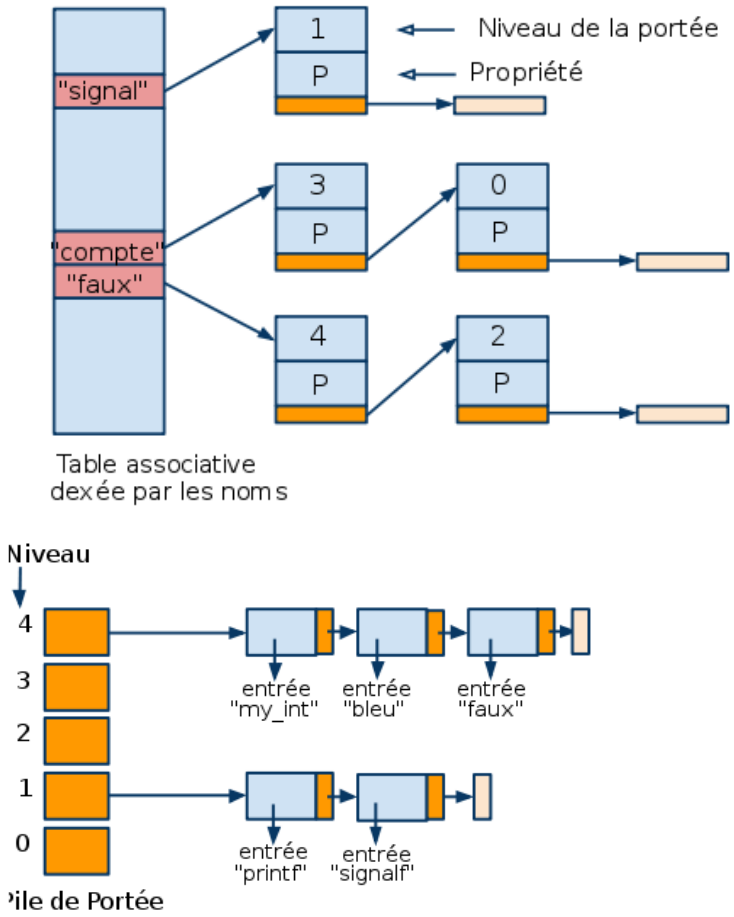


table de n identificateurs.

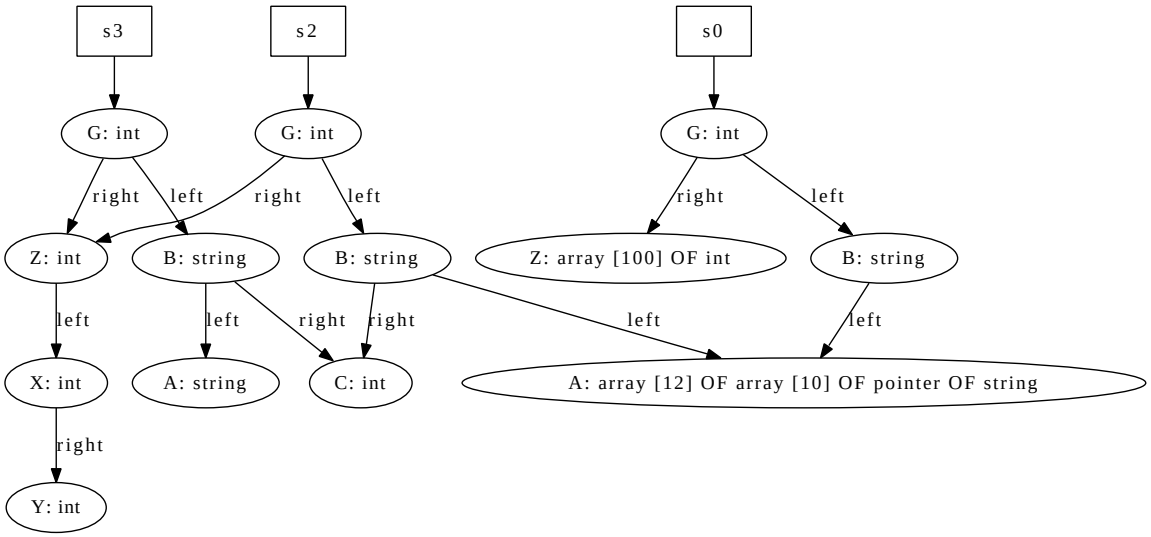
Pour palier à cet inconvénient, on peut rajouter une pile de portées, qui pointe sur toutes les déclarations d'une même portée.
Inconvénient : place plus importante en mémoire.

5.4.3 Une pile d'arbres de recherche persistants :

On empile un nouvel arbre de recherche, à chaque fois que l'on rentre dans une nouvelle portée. Chaque arbre empilé représente une portée ; l'arbre possède des pointeurs vers les identificateurs des portées englobantes qui n'ont pas été redéfinis.
Avantage : recherche et ajout en temps moyen $O(log(n))$.

Implémentation des arbres persistants :

```
fonction ajouter( x : nom, t : type, u : Arbre ) : Arbre
début
  si ( u==NULL)
     retourner new Arbre( x, t, NULL, NULL ) ;
```




```
  sinon si ( x < u.nom)
     retourner new Arbre(u.nom, u.type,
                                ajouter( x, t, u.gauche), u.droit) ;
 sinon si ( x > u.nom)
     retourner new Arbre(u.nom, u.type, u.gauche,
                                 ajouter(x, t, u.droit)) ;
 sinon si ( x==u.nom)
     retourner new Arbre(u.nom, u.type, u.gauche, u.droit) ;
fin
```

Exemple : Soit le bloc de code suivant :

```
{  /* début de portée s0 */
  var
  G: int;
  B: string;
  Z: array [100] of int;
  A: array [12] of array [10] of pointer of string;
```

```
  if (A<5) then { /* début de portée s1 */
     var
     T: int;

     T=-3;
     B=B+"azertyuiop";
    } /* fin de portée s1 */
  else { /* début de portée s2 */
    var
    C: int;
    X: int;
    Z: int;
    Y: int;

    X=0;
    while (X < 100) do { /* début de portée s3 */
    var
         A: string;

         X=X+1;
         A=A+"qsdfghjklm";
      }  /* fin de portée s3 */
  } /* fin de portée s2 */
  A = 8;
}
```

Chapitre 6

Types, vérification de type

6.1 Introduction

Les types des sous-expressions conditionnent le résultat d'une expression. Par exemple l'expression "20"+10 peut donner 30 ou 2010 ou encore une erreur. Le typage va permettre d'une part de rejeter les programmes absurdes, d'autre part d'effectuer une traduction correcte des programmes non-absurdes en langage machine.
Chaque langage possède son système de typage. En se référant à ce système de typage, le compilateur peut contrôler que chaque utilisation de variable ou de fonction est conforme au type attendu. Le compilateur peut aussi inférer le type d'une expression, c'est-à-dire déduire son type d'après le contexte.

6.2 Typage statique, typage dynamique

Si le type d'une variable ou d'une expression est déterminé lors de la compilation : on parle de *typage statique*. C'est le coeur du compilateur qui a pour rôle de déterminer les types. Cela peut être fait au moment de la création de l'arbre de syntaxe abstraite. L'avantage est alors qu'il n'y aura pas de création d'arbre si le typage est incorrect.

Si le type d'une variable est déterminé lors de l'exécution : on parle de *typage dynamique*. Certains contrôles, comme le fait que les indices d'un tableau soient corrects, c'est-à-dire ne dépassent pas les bornes, ne peuvent être faits que dynamiquement.

En java, par exemple, on a un mélange de contrôle statique et de contrôle

dynamique. On dira qu'un système de typage est sain, s'il assure que toutes les valeurs utilisées à l'exécution sont bien du type déterminé statiquement. S'il y a une relation d'ordre entre les types, il faut que les valeurs à l'exécution soit des sous-types des types déterminés à la compilation.

Un langage à *typage statique* peut rejeter des programmes qui aurait qui aurait pu être exécutés sans erreur :

```
if (test)
  return 42 + 75;
else
  return 42 + "ab";
```

Si le test est toujours faux, le programme peut s'exécuter.
Mais si le contrôle de type est statique, il est rejeté.

Un langage à *typage dynamique* implique de faire des tests unitaires, qui testent les exécutions possibles du programme. Mais il est impossible est trop coûteux de faire des tests exhaustifs.

6.3 Sûreté de typage

Définition La sûreté de typage est la capacité du langage à rejeter les programmes absurdes.

Un langage est sûr ou **fortement typé** si tout typage incorrect entraîne la violation d'une règle de typage et donc provoque une erreur (à la compilation ou à l'éxecution).

langage fortement typé ⇒ programmes sans erreur de type.

A contrario, un langage est **faiblement typé** si le comportement du programme n'est plus spécifié en cas de typage incorrect.

Remarque : il existe différentes définitions de **fortement typé**. Pour certains, un langage est **fortement typé** si toutes les variables sont typées et aucune conversion implicite de type n'est possible.

Attention : il n'y a pas de lien entre le fait que le typage soit statique ou dynamique et le fait que le langage soit fortement ou faiblement typé.
- langage C : typage static / faiblement typé
- langage PhP, JavaScript : typage dynamique / faiblement typé

- langages ML, Haskell : typage static / fortement typés
- langages Python, Lisp : typage dynamique / fortement typés

Exemples de langages faiblement typés :

```
var x:= 5;
var y:="37";
var z:= x+y;
```

- en VisualBasic : "37" -> $(int)37$, donc z vaut 42
- en JavaScript : 5 -> "5", donc z vaut "537"
Que se passerait-t-il si $y =$ "ab" ? En JavaScript : z vaut NaN de type `number`.

```
int x= 5;
char y[]="37";
char *z=x+y;
```

- en C : z pointe 5 caractères après le début de y, donc le résultat est indéfini.

Le risque en essayant d'avoir des langages trés fortement typés est de rejeter également un certain nombre de programmes non-absurdes. Le but recherché est d'avoir des langages relativement fortement typés, tout en restant assez expressifs, c'est-à-dire ne rejetant pas trop de programme corrects.

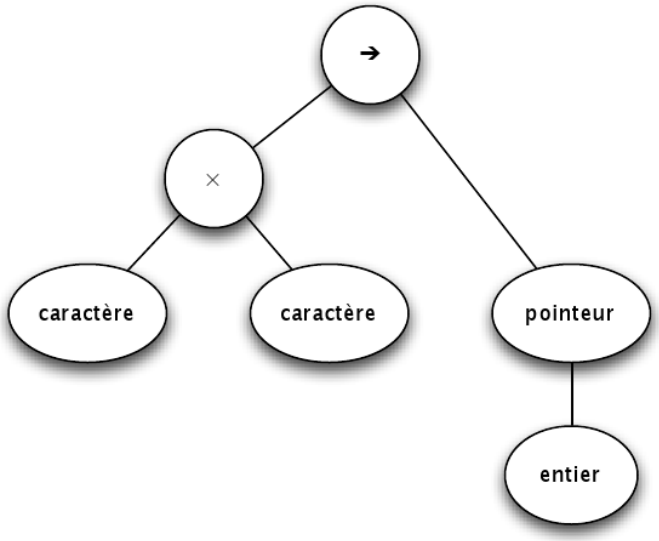
6.4 Représentation des types

Le contrôleur de types utilise un **système de typage**. C'est la donnée de deux choses :
- une représentation des types (types de bases et types complexes),
- un ensemble de règles de typage, premettant de savoir si les expressions sont bien typées.

le contrôleur utilise le système de typage pour contrôler les types lors d'une instruction, ou pour inférer (déduire) le type d'une expression.

6.4.1 Construction des types

On construit des types complexes en appliquant des constructeurs de types à des types simples (types de bases et types nommés).



1. types de base (booléen, caractère, entier, réel), void, error,
2. types nommés introduits par le programmeur (typedef, classes),
3. **constructeurs de types :**
   - tableaux : $array(I, T)$ où I est un ensemble d'indices
   - produits de deux types : $T1 \times T2$
   - structures avec champs : $struct(\, name1 \times T1,\, name2 \times T2)$
   - pointeurs vers un type : $pointeur(T1)$
   - fonctions : $T1 \rightarrow T2$

Pour implémenter les types génériques (comme dans les classes génériques en Java par exemple), on rajoute :
4. des **variables de type :** $'a$ (un type $'a$ inconnu).

6.4.2 Expressions de types

Un type complexe est décrit par une *expression de type* faisant intervenir des constructeurs de types et des types de base.
Exemple : $char \times char \rightarrow pointeur(int)$
On peut représenter une expression de type par un graphe. Le graphe est un arbre si l'expression n'utilise pas de types récursifs.

Types d'un langage pseudo C Les types d'un langage pseudo C peuvent être construits en utilisant les règles de grammaire suivantes :

```
Prog -> Decls Expr
Declist -> Decl Declist | Decl
Decl ->  id : Type | Type id (Type)
Type -> char | int | float | Type[] | Type*
Expr -> littéral | integer | real | Expr(Expr) | Expr[Expr] | *Expr
       | Expr PLUS Expr |...
```

Le contrôleur doit procéder à des vérifications de type dans les expressions. Il doit s'assurer que l'usage d'un identificateur de variable ou de fonction est conforme à son type déclaré.
- Lors de la déclaration d'une variable, il faut construire une représentation de son type (arbre) et la stocker dans l'environnement courant.
- Lors de l'utilisation d'une variable dans une expression, il faut comparer le type déclaré pour la variable et le type attendu par l'expression, et déterminer s'ils sont *équivalents*.

Pour cela on a besoin de savoir quand est-ce que deux expressions de type sont *équivalentes*.

6.5 Equivalence des expressions de type

Dans quelle mesure peut-on dire que deux types sont équivalents ?
Deux types sont équivalents s'ils sont construits à l'aide du même constructeur de type (tableau, pointeur,...) avec des paramètres qui sont eux-mêmes des types équivalents.
Solution simple :

```
fonction Equiv ( s , t ) : booléen
début
  si s et t sont le même type de base alors
      retourner vrai
  sinon si s = tableau( s1, s2) et t = tableau( t1, t2)
  alors
      retourner  Equiv( s1, s2)  et  Equiv( t1, t2)
  sinon si (s = s1 x s2 ) et  (t = t1 x  t2 )
  alors
      retourner  Equiv( s1, s2)  et  Equiv( t1, t2)
    . . .
   sinon
     retourner faux
fin
```

Attention : des cycles peuvent apparaître dans la construction des types, ce qui pose problème pour écrire une fonction d'équivalence récursive.
Exemple :
type lien = pointer of cellule ;

type cellule = structure {
info : int ;
suivant : lien ;
}
Dans ce cas il faut introduire un type nommé et ramené l'équivalence des sous-types nommés à l'égalité des noms.

6.6 Coercition, surcharge et polymorphisme

Problème : une même constante peut représenter des types différents en machine. Un même symbole d'opérateur ou fonction peut représenter différnets code exécutables en machine.
Donc le type associé à une expression peut être différent selon le contexte. Et un identificateur peut correspondre non pas à un seul type, mais à un ensemble de types.

6.6.1 Cohercition

Definition : Coercition : conversion implicite (par le compilateur) d'un opérande dans le type attendu par l'expression.

Cela implique une relation d'ordre entre les types. Le compilateur convertira vers un type supérieur. Dans le cas d'une conversion dans l'autre sens (c'est-à-dire vers un type inférieur) il faut que le programmeur utilise un mécanisme de conversion explicite appelé *cast*.
Exemple :
{ a : int ;
b : real ;
x : int ;
y : real ;
x = a + b ;
y = a + b ; }

Inférence de types à l'aide des attributs :

```
E -> E :e1 op E :e2 {: ...
   if ( e1.getType() == INT && e2.getType() == INT )
         RESULT.putType( INT );
```

```
   else if ( e1.getType() == FLOAT || e2.getType() == FLOAT )
        RESULT.putType( FLOAT );
   else RESULT.putType( ERROR );
    :}
Instr -> LeftExpr :e1 ASSIGN Expr :e2 {: ...
     if ( e2.getType() == INT && e1.getType() == FLOAT )
         RESULT.translate();
         RESULT.getCode().getLeft().RealToInt();
     else if e2.getType() == FLOAT && e1.getType() == INT
         RESULT.translate();
         RESULT.getCode().getRight().RealToInt();
     :}

E -> ID :id {: ...
    RESULT.putType( currentEnv.find(id) );
    :}
E -> E :e1 (E :e2) {: ...
    RESULT.putType( currentEnv.find(e1).getRight() );
       {t|∃s, s ∈ e2.types&&s− > t ∈ e1.types}
:}
```

6.6.2 Surcharge

Definition Surcharge d'opérateur ou de fonction : un nom de fonction ou d'opérateur est surchargé, s'il correspond selon le contexte à plusieurs implémentations (codes machine) différentes.

Dans ce cas, plusieurs expressions de type vont être associées au même opérateur ou à la même fonction. Donc les attributs des noms d'opérateurs ou des noms de fonctions seront des ensembles de types. Pour contôler le type d'une expression, il s'agira alors de faire des parcours de l'arbre de syntaxe abstraite de l'expression, jusqu'au moment où l'on pourra selectionné par élimination un type unique dans l'ensemble.
Exemple :
L'opérateur d'addition peut représenter selon le contexte une addition d'entiers, de réels, de complexes, ou une concaténation de chaines.

Problème :
function $f(i, j : int) : real$; int × int -> real
function $f(i, j : int) : int$; int × int -> int
Quel est le type de $f(4, 5)$? ? ? En particulier dand le cas suivant :
i : i n t ;
x : r e a l ;
$foo(4, 5) + x$; → real
$foo(4, 5) + i$; → int

6.6.3 Polymorphisme

Definition Une fonction ou un opérateur est polymorphe si son code peut être exécuter avec des arguments de types différents.

Contrairement à la surcharge, on a donc ici un seul code machine, quelque soit le type d'arguments.
Pour représenter une fonction ou un opérateur polymorphe, on utilisera des variables de type.
Exemples :
– Opérateur & en C
Si X est de type T , alors $\&X$ est de type pointeur vers T.
– Opérateur [ ] en C
Si X est de type $tableau(T, I)$ et k est de type entier, alors $X[k]$ est de type T.

On peut développer des fonctions polymorphes pour son propre compte.
Exemple : longueur d'une liste

```
# letrec long = function
   [ ] -> 0
   | t ::q -> 1 + long(q) ;

long : list 'a -> int
```

Les types d'un langage polymorphes peuvent être construits en utilisant les règles de grammaire suivantes :

```
P -> D ; E
```

D -> D ; D
     | id : Q

Q -> $\forall id.$ Q
    | T
T -> $T \to T$
     | $T \mid T \times T \mid pointeur(T) \mid liste(T) \mid type \mid ID \mid (T)$
E -> $E(E) \mid E, E \mid ID$

Exemple 1 :
deref : $\forall x$. $pointeur(x) \to x$
Vérifier le type :

```
q : pointeur ( pointeur ( entier ) )

   deref ( deref ( q ) )
pointeur(y) = pointeur(pointeur(entier))
  ppcu = {(y, pointeur(entier))}
pointeur(x) = pointeur(entier)
  ppcu = {( x, entier)}
```

Exemple 2 :
first : $\forall x, \forall y$. $x \times y \to x$
$+$: $int \times int \to int$
Inférer le type de :
$fun : p \to (+(firstp)1)$

```
p: x X y
(first p) :  x
1 : int

(first p)+1 : int
x : int
fun : int X y -> int
```

donc on trouve $fun : \forall y$. $int \times y \to x$

Chapitre 7

Génération de code intermédiaire

Le coeur du compilateur, après avoir vérifier les portées et les types, doit traduire l'Arbre de Syntaxe Abstraite en Code Intermédiaire. Ensuite le Back-End du compilateur traduit le Code Intermédiaire en Assembleur.

7.1 Choix d'une Représentation Intermédiaire

Pour traduire efficacement l'Arbre de Syntaxe Abstraite, on a besoin d'une Représentation Intermédiaire (IR) assez expressive afin d'exprimer toutes les constructions syntaxiques.
Pour facilité le travail du Back-End, on a besoin d'une Représentation Intermédiaire pas trop éloignée des machines réelles, afin d'identifier clairement les instructions de l'Assembleur.
On est donc amener à utiliser successivement plusieurs Représentations Intermédiaires, d'un niveau d'abstraction assez élevé au départ, vers un niveau d'abstraction beaucoup plus bas au final.
Les Représentations Intermédiaires comportent en général :

- des opérateurs arithmétiques,
- des opérateurs logiques,
- des étiquettes, des sauts (conditionnels ou non) vers des étiquettes,
- des accès mémoire (en lecture ou en écriture),
- des déplacements de données d'un registre dans un autre,
- des appels de fonctions.

Les critères de choix d'une Représentation Intermédiaire sont

- la facilité de génération,

– la facilité de manipulation,
– la taille du code produit,
– le pouvoir d'expression.

On commencera par une **Représentation Intermédiaire structurée** sous forme d'arbres ou de dags (sauts), pour aboutir à une **Représentation Intermédiaire linéaire** (pseudo-code assembleur pour machine abstraite). Parallèlement à ces Représentations Intermédiaires du code, on utilise en général un graphe de flot de contrôle, reflétant les exécutions possibles du programme et permettant diverses optimisations.

7.2 Arbres de Code Intermédiaire

Un **arbre de Code Intermédiaire** comportera des noeuds qui peuvent être de type `Stm` (pour les instructions) ou de type `Exp` (pour les expressions). A cela s'ajoute des noeuds de type `Label` pour étiqueter diférents points du programme et des noeuds de type `Temp` pour représenter les registres de calcul.

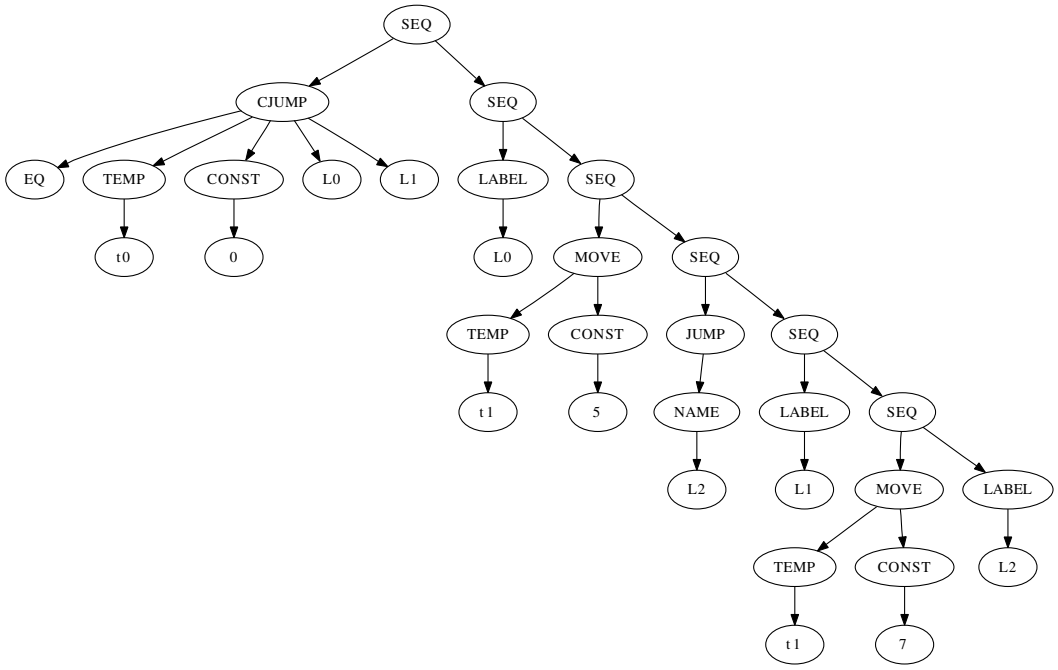
Les noeuds de type `Exp` (sous-classes) sont les suivants :

```
CONST  (int value)
TEMP   (Temp temp)
NAME   (Label label)    // pour un nom de fonction, son étiquette indique
                        // l'adresse  mémoire de la chaîne de caratère
BINOP  (EnumOp binop, Exp left, Exp Right)
MEM    (Exp exp)           // accès en lect./ecrit. selon contexte
CALL   (Exp func, Explist args)
ESEQ    (Stm stm, Exp exp)  // instructions ayant une valeur associée
```

Les noeuds de type `Stm` (sous-classes) sont les suivants :

```
MOVE   (Exp dst, Exp src)   // ecriture dans un registre ou en mémoire
EXP    (Exp exp)            // converti exp en Stm
JUMP   (Exp label)
CJUMP  (EnumRel relop, Exp left, Exp right, Label iftrue, Label iffalse)
SEQ    (Stm left, Stm right)  // liste d'instructions
LABEL  (Label label)          // pose une étiduqette
```

Exemple 1. Une expression $a = x + 2 * y$ sera traduite en :




```
BINOP( PLUS, TEMP(x),
             ESEQ(
                   MOVE( TEMP(t0), BINOP( MULT, CONST(2), TEMP(y))),
                   TEMP(t0)))
```

Exemple 2. Une expression $a[e]$ sera traduite en

```
MEM ( BINOP ( PLUS , MEM( CodeInterm(a)) , WORDMUL( CodeInterm(e))))
```

où $CodeInterm(a)$ (resp. $CodeInterm(e)$) désigne l'arbre de code intermédiaire correspondant à l'expression a (resp. e), et $WORDMUL$ désigne la multiplication (de e) par une constante représentant la taille d'un élément de la pile utilisée pour l'implémentation du compilateur.

Exemple 3. Une instruction conditionnelle : if($t0 == 0$) then $t1 = 5$; else $t1 = 7$; est traduite par des sauts avec trois étidquettes :

```
SEQ( CJUMP ( EQ , TEMP(t1) , CONST(0) , L0 , L1 ),
  SEQ( LABEL ( L0 ),
       SEQ( MOVE( TEMP(t1), CONST(5)),
         SEQ( JUMP ( L2 ),
           SEQ( LABEL ( L1 ),
             SEQ( MOVE( TEMP(t1), CONST(7)),
                 SEQ( LABEL( L2 ))))))))
```

Exemple 4. Une déclaration

```
typedef int arrtype[3];
arrtype array1 = { 0, 0, 0 };
```

est traduite en

```
ESEQ (
  SEQ( MOVE ( TEMP t, CALL ( malloc , WORDMUL( CONST(3)))),
    SEQ( MOVE ( MEM ( BINOP ( PLUS, TEMP t, ( WORDMUL( CONST(0)), CONST 0)))),
      SEQ( MOVE ( MEM ( BINOP ( PLUS, TEMP t, (WORDMUL( CONST(1)), CONST 0)))),
         MOVE( MEM ( BINOP ( PLUS, TEMP t, ( WORDMUL( CONST(2)), CONST 0)))))),
  TEMP t )
```

Remarque :
- L'adressage mémoire est explicité :
lecture : `MOVE( t, MEM(a))` ,
écriture : `MOVE( MEM(a), t)`.
- L'appel de fonction se distingue des autres expressions :
`MOVE( t, CALL( NAME(f), e))`.

7.2.1 Arbres canoniques

But : linéarisation du code.
L'arbre de Code Intermédiaire doit être réécrit sous la forme d'une liste d'arbres canoniques sans `ESEQ` qui pourra ensuite être traduite en Code trois adresses.

1. Faire remonter les noeuds `ESEQ` vers la racine de l'arbre de Code Intermédiaire, afin de les éliminer.
   Attention : les `ESEQ` peuvent faire des effets de bord $\Rightarrow$ si les sous-expressions sont evaluées dans un ordre différent, cela peut donner des résultats différents.
2. Déplacer les appels de fonctions pour qu'ils figurent en tête des expressions : `EXP(CALL(f,...))` ou `MOVE(TEMP t, CALL(f,...))`.
3. Dans la plupart des machines, pour les tests conditionnels, si le test est faux on continue à l'instruction suivante -> réorganiser les `CJUMP` de sorte qu'ils soient immédiatement suivis par la partie à exécutée si le test est faux.

Exemple 1 : $t1 = x + 2 * y$

```
MOVE( t1, BINOP( PLUS, TEMP(x),
                 ESEQ( MOVE( TEMP(t0), BINOP( MULT, CONST(2), TEMP(y))),
                       TEMP(t0))))
```

Cet arbre sera réécrit en :

```
MOVE( TEMP(t1), ESEQ( MOVE( TEMP(t0), BINOP( MULT, CONST(2), TEMP(y))),
                      BINOP( PLUS, TEMP(x), TEMP(t0) )))
```

Puis réécrit en :

```
SEQ( MOVE( TEMP(t0), BINOP( MULT, CONST(2), TEMP(y))),
       MOVE( TEMP(t1), BINOP( PLUS, TEMP(x), TEMP(t0) )))
```

Puis traduit en Code 3 adresses :

```
t0 = 2 * y;
t1 = x + t0;
```

Exemple 2 : $a = f(x + 2 * y)$

```
 MOVE( TEMP(a),
       CALL(f, ESEQ( SEQ( MOVE( TEMP(t0), BINOP( MULT, CONST(2), TEMP(y))),
                          MOVE( TEMP(t1), BINOP( PLUS, TEMP(x), TEMP(t0) ))),
                     TEMP(t1))))
```

Cet arbre sera réécrit en :

```
MOVE( TEMP(a),
      ESEQ( SEQ( MOVE( TEMP(t0), BINOP( MULT, CONST(2), TEMP(y))),
                 MOVE( TEMP(t1), BINOP( PLUS, TEMP(x), TEMP(t0) ))),
            CALL(f, TEMP(t1))))
```

Puis réécrit en :

```
SEQ( MOVE( TEMP(t0), BINOP( MULT, CONST(2), TEMP(y))),
   SEQ( MOVE( TEMP(t1), BINOP( PLUS, TEMP(x), TEMP(t0) ))
         MOVE( TEMP(a), CALL(f, TEMP(t1)))))
```

Puis traduit en Code 3 adresses :

```
t0 = 2 * y;
t1 = x + t0;
a = call( f, t1);
```

Principales transformations, avec s et e1 qui commutent :

le code intermédiaire :	est remplacé par :
MEM(ESEQ(s,e))	ESEQ(s,MEM(e))
MOVE(TEMP t, ESEQ(s,e))	SEQ(s, MOVE(TEMP t,e))
BINOP(op, ESEQ(s, e), e1)	ESEQ(s, BINOP(op, e, e1))
CJUMP(op, ESEQ(s, e), e1, L1, L2)	SEQ(s , CJUMP(op, e, e1, L1, L2))
BINOP(op, e1, ESEQ(s, e))	ESEQ(s, BINOP(op, e1, e))
CJUMP(op, e1, ESEQ(s, e), L1, L2)	SEQ(s , CJUMP(op, e1, e, L1, L2))

Si s et e1 ne commutent pas :

le code intermédiaire :	est remplacé par :
BINOP(op, e1, ESEQ(s, e))	ESEQ(MOVE(TEMP t, e1), ESEQ(s, BINOP(op, TEMP t, e)))
CJUMP(op, e1, ESEQ(s, e), L1, L2)	SEQ(MOVE(TEMP t, e1), SEQ(s, CJUMP(op, TEMP t, e, L1, L2)))

7.3 Code trois adresses

Assembleur de haut niveau comportant des sauts, des étiquettes, et où chaque opérateur a au plus deux opérandes.
On regroupe dans des blocs basics les parties de code qui ne contiennent ni saut ni label -> nécessairement exécutés séquentiellement, sans interruptions.

Exemple 1. En supposant a, x et y déjà dans des registres, une instruction $a = x + 2 * y + 1$; sera traduite en :

```
t0 = 2 * y ;
t1 = x + t0 ;
a = t1 + 1 ;
```

Exemple 2. Une instruction $x = a[i]$; sera traduite en

```
t0 = a;
t1 = w * i; // w est la taille d'un élément de pile
x = t0 + t1;
```

Exemple 3. Une instruction conditionnelle if ($t0! = 0$) then $t1 = 5$; else $t1 = 7$; sera traduite en :

```
Cjump( NEQ, t0, 0, L1, L2);
L1:
  t1 = 5;
L2:
  t1 = 7;
```

Remarque :
- On obtient un mélange de code séquentiel et de sauts,
- On suppose lors de cette traduction, que l'on dispose d'un nombre infini de registres.

On utilisera donc un processeur séquentiel avec des registres et des accès mémoire.
- Le calcul du nombre de registres nécessaires se fera plus tard,
- L'utilisation effective des registres du processeur se fera plus tard.
- La traduction des appel de fonctions se fera plus tard(dépend du processeur cible).

Chapitre 8

Du code intermédiaire vers le code optimisé

L'optimalité du code est indécidable, puisque le problème de savoir si deux programmes ont le même comportement (terminent ou non) sur toutes les entrées est indécidable.
Plusieurs sortes d'optimisations :

- optimiations locales : à l'intérieur des blocs
- optimiations globales : sur l'ensemble du graphes de flot de contrôle, c'est -à-dire sur l'ensemble des blocs.

Il existe aussi des optimisations dépendantes du code cible, qui seront faites par le back-end.

8.1 Optimiations locales

- Elimination de sous-expressions communes.
- Simplifications arithmétiques ($x = x + 0;$).
- Reduction de force (remplacer des opérations par d'autres moins coûteuses).
- Propagation des constantes ($a = 2;\ x = y + a;$)
- Elimination de code mort (code inutile).
- Propagation des copies.

8.2 Optimiations globales

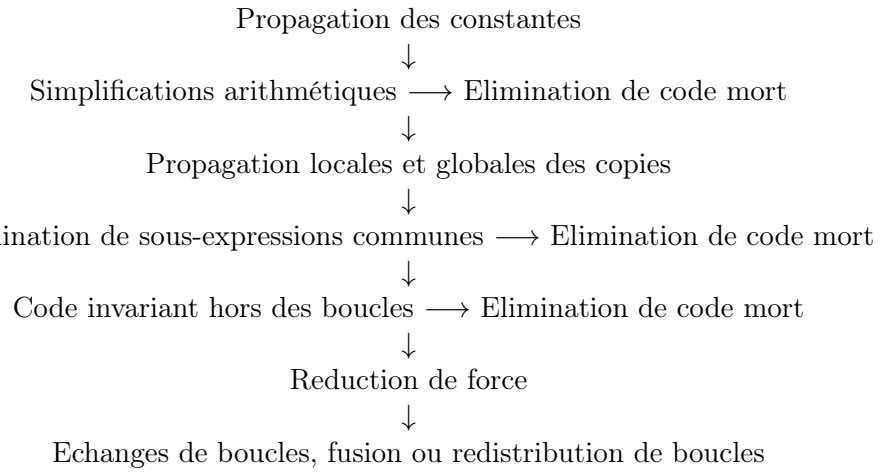
- Déplacement de code (code invariant hors des boucles, échanges de boucles).

- Elimination globale de code mort.
- Propagation globale des constantes.
- Elimination des redondances (calcul de valeurs déjà connues).

8.3 Ordre des optimisations

L'ordre dans lequel on effectue les optimisations est important car elles interagissent.
Exemple :

```
x=5;
...
if (x<10) then ... // propagation des cstes -> elimination code mort
```



8.4 Analyse du flot de données

Il s'agit d'un raisonnement statique (à la compilation) sur des flots dynamiques de données (qui seront obtenus à l'exécution). L'analyse statique permet d'obtenir des informations qui sont ensuite utilisées pour optimiser le programme. Le programme est d'abord représenté sous forme d'un graphe, le **graphe de flot de contrôle**, indépendant du langage source ou du langage cible du compilateur.

Les questions auxquelles doit répondre l'analyse du flot de données sont :

– Quelles affectations de variables "atteignent" un certain point du programme (i.e. sont encore valides en ce point, quelque soit l'exécution) ?
– Quelles variables sont lues/modifiées dans un bloc donné ?
– Est-ce que une occurrence de la variable x est la dernière du programme ?

L'analyse du graphe de flot de données consiste à :
– écrire un système d'équations dont les inconnues sont des ensembles de variables ou des ensembles d'expressions du programme,
– résoudre ces équations en recherchant le plus petit ou plus grand point fixe (programmation dynamique).

Les solutions de ces équations sont ensuite utilisées pour en déduire des optimisations (suppression de code mort, de code redondant,...).

L'analyse peut se faire de deux façons :
- Analyse avant : on détermine les propriétés d'un bloc en fonction de ses prédecesseurs,
- Analyse arrière : on détermine les propriétés d'un bloc en fonction de ses succésseurs.

8.4.1 Graphe de flot de contrôle

Les sommets du **graphe de flot de contrôle** sont des blocs d'instructions et les arcs représentent le flot des données au cours des différentes exécutions possibles.

Découpage du code linéarisé en blocs :
Toute étiquette marque le début d'un bloc. Tout saut ou branchement marque la fin d'un bloc. Toute instruction suivant un branchement ou un saut marque le début d'un autre bloc.

Un arc relie deux sommets du graphe de contrôle si le contrôle peut passer du premier sommet au deuxième au cours d'une exécution particulière du programme.

Exemple :

```
B1 :
  a = 0
B2 :
  LABEL L1
```

```
  b = a+1
  c=c+b
  a=a *2
  if a<n goto L1 else goto L2
B3 :
  LABEL L2
  print c
```

Les arcs sont :

```
B1 -> B2
B2 -> B2
B2 -> B3
```

8.4.2 Analyse des variables vivantes

Une variable est vivante à la sortie d'un bloc `B` si elle est utilisée par un bloc que l'on peut atteindre depuis `B`.

On notera $succ(B)$ l'ensemble des blocs successeurs du bloc `B` dans le graphe de flot de contrôle. Il s'agira d'une analyse arrière, car le calcul des ensembles de variables pour un bloc `B` se fera en fonction des valeurs obtenues pour les successeurs de `B`.
On définit d'abord quatre ensembles de variables.
$Out(B)$: variables vivantes en sortie de B
$In(B)$: variables vivantes en entrée de B
$Def(B)$: variables qui figurent dans un membre gauche dans B
$Use(B)$: variables qui figurent dans un membre droit dans B avant d'être définies (i.e. de figurer dans un membre gauche)

Les équations reliant ces ensembles sont les suivantes :

$$In(B) = Use(B) \cup (Out(B) - Def(B))$$

$$Out(B) = \cup_{B' \in Succ(B)} In(B')$$

Algo :

1. Déterminer Use et Def pour chaque bloc.

2. Initialiser Out (et In) à $\emptyset$ pour chaque bloc.
3. Appliquer les équations -> nouvelles valeurs des ensembles In et Out.
4. Tant que In et Out augmentent, retourner en (3).

L'algorithme termine puisque l'ensemble des variables du programme est fini.

On effectue i boucles de calcul. Si $Out_i(b)$ désigne l'ensemble $Out(B)$ calculé lors de la $i$ème boucle, on a $Out_0(B) = \emptyset$. A la boucle $i+1$, on a :

$$In_{i+1}(B) = Use(B) \cup (Out_i(B) - Def(B))$$

$$Out_{i+1}(B) = \cup_{B' \in Succ(B)} In_{i+1}(B')$$

Exemple

```
L1 :
t1=a
L2 :
t2=b
t3=a+b
if (t1 > t2) goto L3 else goto L4
L3 :
t2=t2+b
t3=t3+c
if (t2 > t3) goto L5 else goto L2
L4 :
t4=t3 * t3
t1=t4+t3
goto L1
L5 :
print(t1, t2, t 3)
```

8.4.3 Optimisation

- Allocation de registres : si une variable n'est pas vivante en entrée d'un bloc on peut réassigner son registre.

- Elimination de code mort dans un bloc :
éliminer les affectations aux variables qui ne sont pas vivantes en sortie de bloc ni lues ultérieurement dans le bloc.

8.4.4 Propagation des copies

Pour pouvoir propager les les copies, on a besoin de connaître l'ensemble $In(B)$ de toutes les instructions du type $x = y$ qui atteignent l'entrée d'un bloc B et l'ensemble $Out(B)$ de toutes les instructions du type $x = y$ qui atteignent la sortie du bloc.
Une copie $x = y$ est "produite" par B si elle appartient au bloc B et il n'y a pas d'affectation ultérieure à y dans B. L'ensemble des copies produites par B est $Prod(B)$.
Une copie $x = y$ est "supprimée" par B si elle n'appartient pas au bloc B et x ou y sont définies dans B. L'ensemble des copies supprimées par B est $Supp(B)$.

Les équations reliant ces ensembles sont les suivantes :

$In(B_1) = \emptyset$,

pour $B \neq B_1$, $In(B) = \cap_{B' \in Pred(B)} Out(B')$

$Out(B) = Prod(B) \cup (In(B) - Supp(B))$

On effectue alors une analyse avant et une recherche de plus grand point fixe.
Si le bloc B contient une utilisation de x, si la copie $x = y$ est dans $In(B)$ et aucune affectation à x ou y n'est faite entre le début du bloc et l'utilisation de x, alors on peut remplacer x par y dans B.
Si on a pu remplacer toutes les utilisations de x par y, alors on peut supprimer la copie $x = y$.

Exemple :

```
L1 :
t1=a+b
x=y
if (t1 > 0) goto L2 else goto L3
L2 :
t2=b+1
y=2*t2
goto L5
L3 :
```

```
x=z
L4:
t2=x-1
L5:
t1=a+x
```

En résolvant les équations, on trouve qu'aucune des deux copies n'atteint B_5. Elles ne peuvent donc pas être propagées.

8.5 Optimisation des boucles

8.5.1 Définition des boucles et des dominants

Un ensemble $\mathcal{B}$ de blocs est une **boucle** de tête B_h ssi cet ensemble de blocs vérifie :

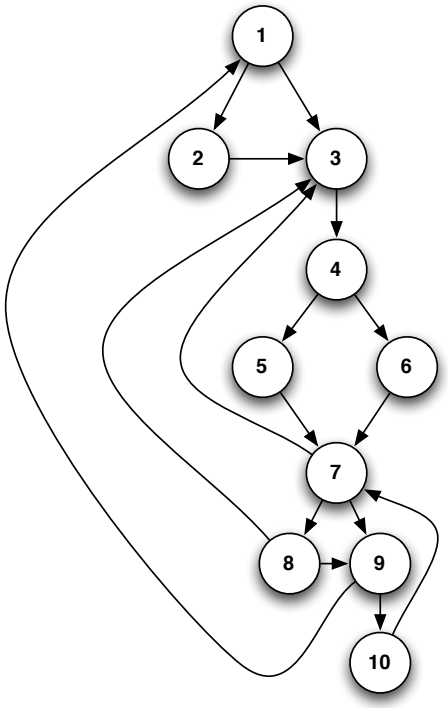
- pour tout bloc B de $\mathcal{B}$, il existe un chemin dans le graphe de contrôle de flot, allant de B_h à B, dont tous les sommets sont dans la boucle et un chemin allant de B à B_h, dont tous les sommets sont dans la boucle,
- si il existe un arc $B \rightarrow B'$ du graphe de contrôle de flot, avec B' dans la boucle et B en dehors de la boucle, alors B' est égal à B_h (Un seul point d'entrée dans la boucle).

Un sommet B du graphe de contrôle de flot **domine** le sommet B' ssi tout chemin du graphe de contrôle de flot, partant du bloc initial et arrivant à B', passe nécessairement par B.
Par définition :
- tout bloc se domine lui-même
- la tête de boucle domine les autres blocs de la boucle.

L'ensemble des dominants du bloc B vérifie :
$Dom(B) = \{B\} \cup \cap_{B' \in Pred(B)} Dom(B')$

Algorithme de calcul des dominants
Soient B_1, B_2, ... B_n, les blocs du graphe de contrôle de flot :

1. initialiser $Dom(B_1)$ à $\{B_1\}$ et pour tout $i > 1$, initialiser $Dom(B_i)$ à l'ensemble des blocs,
2. pour tout i de 2 à n, calculer :
   $Dom(B_i) = \{B_i\} \cup \cap_{B_j \in Pred(B)} Dom(B_j)$



3. tant que l'un des ensemble $Dom(B_i)$ change, reprendre en (2).

Dans l'exemple ci-dessus, les boucles sont :
$\{7, 8, 9, 10\}$ et $\{3, 4, 5, 6, 7, 8, 9, 10\}$

8.5.2 Instructions invariantes dans une boucle

Ce sont les instructions qui calculent les mêmes valeurs depuis l'entrée du contrôle dans la boucle jusqu'à la sortie de la boucle.
Une des optimisations importantes des boucles consiste à détecter les instructions invariantes, afin de les sortir de la boucle.

Algorithme de calcul des instructions invariantes

1. Marquer comme invariantes les instructions dont les opérandes sont des constantes ou ont toutes leurs definitions visibles en dehors de la boucle.
2. Répéter l'étape 3 jusqu'à aucune nouvelle instruction invariante ne soit marquée.
3. Marquer comme invariantes les instructions dont les opérandes vérifient une des conditions suivantes :
   . être une constante
   . avoir toutes les définitions visibles à l'extérieur de la boucle
   . avoir exactement une définition visible à l'intérieur de la boucle et cette définition est marquée.

Algorithme de déplacement de code hors de la boucle
Condition pour déplacer une instruction invariante $(i) : x = y + z$ en dehors de la boucle :

1. le bloc contenant (i) domine tous les blocs de sortie de boucle,
2. la boucle ne contient aucune autre définition de x,
3. aucune définition de x autre que (i) (c'est-à-dire extérieure à la boucle) n'atteint une utilisation de x dans la boucle.

On peut alors déplacer l'instruction (i) dans un bloc "preheader", c'est-à-dire un bloc placé immédiatement avant la boucle.

8.5.3 Remplacement des variables d'induction dans une boucle

Le but de cette optimisation est d'avoir moins de calculs effectués à chaque passage dans la boucle, ou alors des calculs moins coûteux : réduction de force.

On distingue deux types de variables d'induction :

- **variables d'induction de base** = variables incrémentées ou décrémentées d'une quantité constante à chaque passage dans la boucle : $i = i \pm c$, où c est un invariant de boucle,

- **variables d'induction dérivées** = variables obtenues comme somme ou produit (resp. soustraction ou division) d'une autre variable d'induction et d'un invariant de boucle :
  $k = a * j$ ou $k = j + b$, où a et b sont des invariants de boucle et j une variable d'induction,
  à condition que :
  - la seule définition de j qui atteint k est dans la boucle,
  - si j est elle-même dérivée de i, il n'y a pas de définition de i entre la définition de j et celle de k.

Algorithme de remplacement des variables d'induction dérivées :
Soit $k = a * j$ (resp. $k = j + b$) une variable d'induction dérivée,

1. remplacer la définition de k par $k = k'$, où k' est une nouvelle variable,
2. si j est une variable d'induction de base, ajouter à la fin du bloc où j est initialisée, une instruction de prétraitement $k' = a * j_0$ (resp. $k' = j_0 + b$) avec j_0 la valeur initiale de j,
3. ajouter après toute instruction $j = j \pm c$ une instruction $k' = k' \pm a * c$ (resp. $k' = k' \pm c$),

remarque : pour $k' = k' \pm a * c$: faire le calcul de la constante $c_0 = a * c$ dans un bloc de prétraitement, puis écrire $k' = k' \pm c_0$,

4. remplacer toute utilisation de k par k'.

A la suite de cet algorithme, on peut éventuellement éliminer du code mort dans les cas suivant :

1. si j (dont k est dérivée) n'est utilisée dans la boucle que dans une définition d'elle-même et si j n'appartient pas aux variables vivantes des blocs de sortie de boucle, alors on peut supprimer j,
2. si j (dont k est dérivée) est utilisée dans la boucle, dans une condition de saut : `if (j > N) goto L1` , remplacer ce saut par
   `if (k > a*N) goto L1` , si a positif, ou par
   `if (k < a*N) goto L1` , si a négatif,
   (resp. par `if (k > N±b) goto L1` , si $k = j \pm b$).

Exemple : On rappelle qu'une instruction $x = a[k]$; est traduite en

```
t0 = a
t1 = w * k   //  w est la taille d'un élément de pile
x = t0 + t1
```

Dans l'exemple suivant `w=4` .

```
L1 :
i=m-1
j=n
t0=a
t1=4*n
v=t0+t1     // équivaut à v=a[n]
L2 :
i=i+1
t2=4*i
t3=t0+t2   //  équivaut à t3=a[i]
y=2*t2
if (t3 < v) goto L2  else  goto L3
L3 :
j=j-1
t4=4*j
t5=t0+t4   //  équivaut à t5=a[j]
if (t5 > v) goto L3  else  goto L4
```

```
L4:
if (j < i) goto L2  else  goto L5
L5:
print(t3,t5)
```

Chapitre 9

Allocation de registres

La valeur des variables doit être stockée depuis la définition de la variable jusqu'à ses différentes utilisations.
Deux possibilités : stocker la valeur en mémoire (sur la pile) ou stocker la valeur dans un registre de la machine.
Problème : le nombre de registres est limité.
Il faut savoir s'il est possible de stocker toutes les variables dans des registres, car c'est plus avantageux en temps d'accès, ou si il est nécessaire de mettre certaines variables sur la pile. Ces décisions ont un fort impacte sur le temps d'exécution du programme.
On utilise les informations de l'analyse du flot de donnees :
. si une variable n'est pas vivantes à la fin d'un bloc, alors on peut libérer son registre,
. si deux variables sont vivantes en même temps, on ne peut pas les stocker dans le me registres, il faudra deux registres différents.

9.1 Graphes di'nterférence des variables

Les sommets du graphes sont les variables du programme.
Il y a une arrête entre deux variables si elles sont simultanément vivantes en sortie d'une instruction, on dit qu'elles interfèrent.
Pour savoir le nombre minimal de registres nécessaires pour exécuter le programme, on utilise une heuristique de coloration de graphe.

Rappel : savoir si un graphe est coloriable en k couleurs avec $k > 2$, et de telle sorte que deux voisins aient toujours des couleurs différentes, est NP-complet.

On n'a donc pas d'algorithmes, juste des heuristiques.
Principe : si il existe un sommet x du graphe d'interférence G tel que x a un degré strictement plus petit que k, alors si $G - \{x\}$ est k-coloriable, G est k-coloriable.

9.2 Heuristique de coloriage avec k couleurs

- trouver un sommet de degré strictement plus petit que k,
- enlever ce sommet du graphe,
- récursivement appliquer l'heuristique de coloriage au reste du graphe,
- remettre le sommet supprimé dans le graphe,
- le colorier avec la couleur disponible (il en existe nécessairement une).

Si l'heuristique bloque alors on met en mémoire l'une des variables qui a causé le blocage et on recommence avec un sommet en moins dans le graphe.

Pour construire le graphe d'interférence, ont relie par un arc particulier, appelé arc "MOVE", les variables qui sont des copies l'une de l'autre. Ce type d'arc n'est pas pris en compte dans l'heuristique de coloriage. On essaie cependant de trouver un coloriage qui assigne la même couleur aux variables liées par un arc "MOVE". Si c'est le cas on peut alors fusionner ces deux variables en une seule.

Exemple Soit le bloc suivant, dont les variables vivantes en sortie sont d, j, k.

```
g=M[j+12]
h=k-1
f=g*h
e=M[j+8]
m=M[j+16]
b=M[f]
c=e+8
d=c
k=m+4
j=b
```

Le graphe d'inférence de ce code est coloriable avec 4 couleurs.
Les variables d et c sont identifiables (remplacer tous les c par d).
Les variables j et b sont identifiables (remplacer tous les b par j).

9.3 Variables spillées

Si le nombre maximal de registres est fixé et que le coloriage échoue pour une variable x, il faut stocker x en mémoire (on dit alors que x est une variable "spillée").

- choisir une adresse mémoire m_x pour stocker x,
- pour chaque apparition de x dans le programme, introduire une nouvelle variable x_j :
  si l'instruction utilise x_j, rajouter juste avant $MOVE(x_j, m_x)$,
  (remarque : pour cette instruction $Def = \{x_j\}$ et $Use = \emptyset$)
  si l'instruction définit x_j, rajouter juste après $MOVE(m_x, x_j)$,
  (remarque : pour cette instruction $Def = \emptyset$ et $Use = \{x_j\}$)
- en général, x_j est vivante pendant deux ou trois instructions, donc interfère peu avec les autres variables.

On peut tracer un nouveau graphe d'interférence où le degré des nouvelles varaibles x_j est strictement inférieur au degré de x dans le précédent graphe. On peut donc le colorier avec un plus petit nombre de couleurs.

Exemple Dans l'exemple précédent, si la variable f est spillée, le graphe d'inférence est coloriable en 3 couleurs.

Chapitre 10

Le code compilé

Definition : L'ensemble des structures de données maintenues à l'éxecution (pile, tas, zone de mémoire statique) pour implémenter les concepts de haut niveau est appelé **Environnement d'exécution** ("runtime environment").

Il est nécessaire de tenir compte du futur Environnement d'exécution, lors de la traduction en Code Intermédiaire :

- comment les variables et les fonctions sont-elles stockées en mémoire ?
- comment sont implémentés les appels de fonctions, les passages de paramètres ?
- comment sont implémentés les tableaux ?
- comment est implémentée l'allocation et éventuellement la libération de la mémoire ?

10.1 Organisation de l'espace de travail

Si le langage source n'a pas simultanément des définitions de fonctions imbriquées et des fonctions comme valeur de retour d'autres fonctions, alors les appels de fonctions se comportent comme une pile :
variables locales crées à l'entrée des fonctions
variables locales détruites à la sortie des fonctions.

Le bloc de mémoire du code compilé est séparé en quatre espaces :
1. le Code cible produit
2. la Zone de mémoire statique
3. la Pile des appels de fonctions
4. le Tas

Le bloc de la pile relatif à un appel de fonction est appelé **bloc d'activation** ("activation record" ou "stack frame" en anglais).

Les données dont la durée de vie est incluse dans le bloc d'activation d'une fonction (variables locales) seront allouées en pile ou en registres).
Les variables globales sont en zone de mémoire statique.
Toutes les autres informations (données allouées dynamiquement) sont contenues dans une zone séparée, appelée le tas.

10.2 Enregistrement d'activation

On gère la pile à l'aide de deux pointeurs :
`SP` (Stack Pointer) est le pointeur de sommet de pile (la pile est souvent impémentée en ordre inverse : avancer le pointeur de pile correspond à le décrémenter),
`FP` (Frame Pointer) est le pointeur vers le début des variables locales du bloc d'activation courant.

Exemple : L'expression `BinOp Add E1 E2` correspond au calcul suivant

```
code(E1 ); code(E2 );
PILE[SP - 1] := PILE[SP - 1] + PILE[SP] ;
SP := SP - 1 :
```

Lorsqu'une fonction g appelle $f(a1, ..., an)$, le **bloc d'activation** de f est alloué (dans la pile). Il contient :

1. les paramètres de l'appel $a1, ..., an$ (transmis par g),
2. l'adresse de retour de f (adresse de la première ligne de code à exécuter après le retour de l'enregistrement d'activation de f),
3. le pointeur `FP` de g,
4. l'état machine (sauvegarde des registres de la machine) avant l'appel de f,
5. les variables locales de f,
6. les temporaires utilisés pendant les calculs de f.

Appel de fonction : appel de f par g :

1. g évalue les arguments $a1, ..., an$ de f,
2. g stocke dans le bloc d'activation de f son adresse de retour et son pointeur FP,
3. f sauvegarde l'état machine,
4. f initialise ses variables locales et exécute son code.

Retour de fonction : retour de la fonction f :

1. f place la valeur de retour en suivant de l'enregistrement d'activation de g,
2. f restaure l'état machine,
3. f se branche à l'adresse de retour et efface de la pile son bloc d'activation,
4. g utilise la valeur de retour de f et restaure le pointeur de pile SP.

10.3 Allocation mémoire

- **Allocation statique :**
  on dispose d'un pointeur GP (Global Pointer) sur la zone statique. Une nouvelle adresse de variable globale est obtenue en décalant le pointeur de la zone statique.
  La taille à allouer doit être connue à la compilation (pas d'allocation dynamique dans cette zone).
  Pas de fonctions récursives utilisant cette zone (tous les appels utiliseraient la même adresse).

- **Allocation en pile :**
  Pas de perte d'espace mémoire (les variables locales sont supprimées)
  La taille de la mémoire à libérer (enregistrement d'activation au retour d'une fonction) est connue du compilateur.

- **Allocation dans le tas :** allocation dynamique.
  Le tas est souvent géré comme une suite de blocs mémoire libres ou occupés.
  - Allocation explicite prévue par le programme source (ex : malloc). Dans ce cas la libération de la mémoire doit aussi être explicite.
  - Allocation implicite lorsque le processeur le demande.

Problème du "rebut" : zone allouée dans le tas mais inaccessible. Les langages à forte allocation dynamique (Java, Lisp) pratiquent la récupération du "rebut" par un ramasse-miettes (Garbage collector).

– **passage paramètres :**

1. par valeur : écrire directement les valeurs dans l'enregistrement d'activation de la fonction appelée.
2. par référence (pointeur) : écrire les adresses des paramètres dans l'enregistrement d'activation de la fonction appelée.
3. par copie-restauration (valeur-résultat) :
   à l'appel, la fonction appelante copie les valeurs des paramètres dans l'enregistrement d'activation de la fonction appelée,
   au retour, la fonction appelée copie les nouvelles valeurs des paramètres dans l'enregistrement d'activation de la fonction appelante.
4. par nom : substitution textuelle des expressions passées en argument dans le corps de l'a fonction appelée (implémentée par le passage de l'adresse de l'expression).

10.3.1 En pratique

En java, une classe `Frame.java` implémentera les enregistrements d'activations. Une instance de cette classe sera associé à chaque fonction. Cette classe contiendra plusieurs variables statiques : les pointeurs SP et FP (pointeurs sur la pile), le pointeur GB (pointeur sur le tas), un pointeur RV (return value) vers l'adresse de retour, un entier $WordSize$ donnant la taille d'un mot de pile.

Le back-end du compilateur rajoutera deux morceaux de code à chaque appel de fonction : un "prologue " rajouté au début et un "épilogue" rajouté à la fin.
Le prologue se charge d'aller chercher les paramètres de la fonction, là où la machine cible les attend.
L'épilogue se charge de placer la valeur de retour, là où la machine cible l'attend.