

# Notes du cours de compilation

## Bibliographie incontournable pour apprendre le cours

- Modern Compiler Implementation in Java 2nd Edition, 1998, Andrew W. Appel, Cambridge University Press
- Aho, Sethi, Ullman, Compilers : Principles, Techniques, and Tools, Addison-Wesley, 1986
- Traduction refondue en français Aho, Sethi, Ullman, Compilateurs : principes, techniques et outils, InterÉditions, 1989. Dunod, 2000.

## 1 Introduction

Un compilateur est un logiciel de traduction d'un langage source vers un langage cible. D'ordinaire le langage source est un langage de programmation évolué, comme C++ ou Java par exemple, le langage cible un code machine prévu pour programmer un ordinateur en particulier.

Un compilateur contient plusieurs modules :

- analyseur lexical
- analyseur syntaxique
- analyseur sémantique
- générateur de code intermédiaire
- optimiseur de code
- générateur de code

Il gère une table des symboles, détecte et signale un ensemble d'erreurs à chaque niveau d'analyse et produit le code qui servira à la programmation.

Le compilateur entre dans un processus d'élaboration des programmes et n'est qu'un des rouages permettant de construire un programme. 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

FIGURE 1 – Contexte du compilateur

Les phases et le résultat peuvent être différents d'un compilateur à un autre :

- Interprétation plutôt que compilation proprement dite : Postscript, Shell, HTML

- Production de code portable (bytecode) pour machine virtuelle, et non de code dédié à une machine en particulier : P-code, java, NET
- Langages sources plus ou moins structurés. L'assembleur par exemple est très peu structuré, il présente des instructions machines, des directives, des étiquettes.
- Optimisations plus ou moins poussées
- Analyse des erreurs plus ou moins poussées

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 :  
commentaires, mots réservés, constantes, identificateurs
2. Analyse syntaxique :  
Transforme le code en arbre de syntaxe abstraite
3. Analyse sémantique, génération de code intermédiaire :
  - Types :  
 $\text{PGCD } int \times int \rightarrow int$   
 $\mathbf{a} \text{ } int$   
 $\mathbf{b} \text{ } int$
  - Instructions :  $SUCC(WHILE(TEST(\neq, a, b), IF(TEST(<, a, b), AFF(a, OP(-, a, b)), SUCC(BLOC(VAR(tmp, int), AFF(tmp, a), SUCC(AFF(a, b), AFF(b, tmp))))), RETURN(a))$

## 2 Analyse lexicale

### 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

### 2.2 Expression rationnelle

Soit un alphabet  $A$ , un ensemble rationnel est :

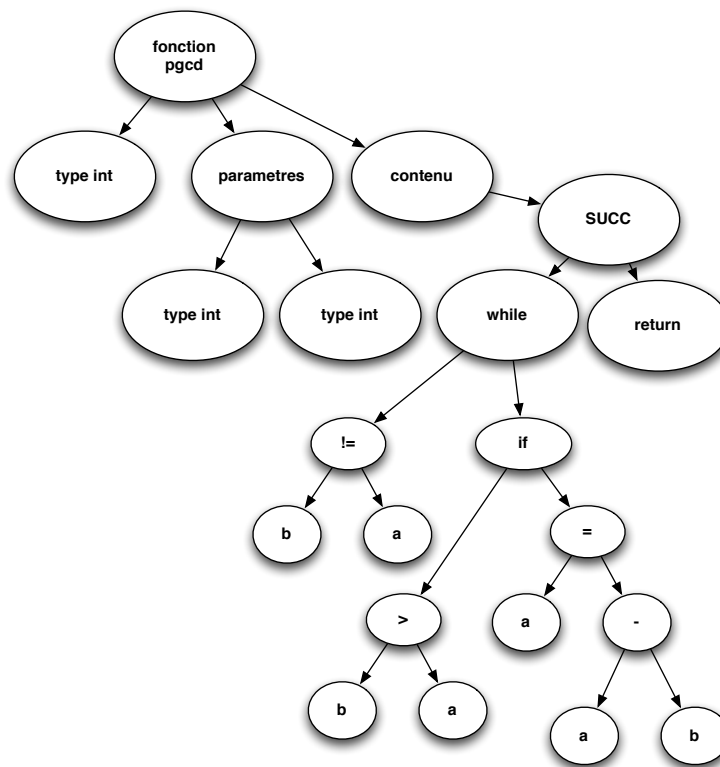


FIGURE 2 – arbre syntaxique

1. Ensemble fini de mots
2. Concaténation d'ensembles rationnels  $R_1R_2$
3. Itération d'ensembles rationnels  $R^*$
4. Union d'ensembles rationnels  $R_1 \cup R_2$

Exemple :

Alphabet = [A-Z] [a-z] [0-9] "\_"

Lettre = [A-Z] [a-z]

Chiffre = [0-9]

Identificateur = Lettre (Lettre  $\cup$  Chiffre  $\cup$  "\_")\*

Entier = {0}  $\cup$  [1-9] (Chiffre)\*

## 2.3 Automate

Un analyseur lexical s'implémente à l'aide d'un automate

## 2.4 JFLEX

Construit un automate qui applique les actions à effectuer sur chaque entité décrite par les expressions rationnelles.

Fichier flex  $\rightarrow$  JFlex  $\rightarrow$  Lexer.java  $\rightarrow$  Java  $\rightarrow$  Lexer.class

```
java JFlex pseudocode.jflex
```

```
javac Lexer.java
```

```
java Lexer programme.pseudocode
```

Un 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]
```

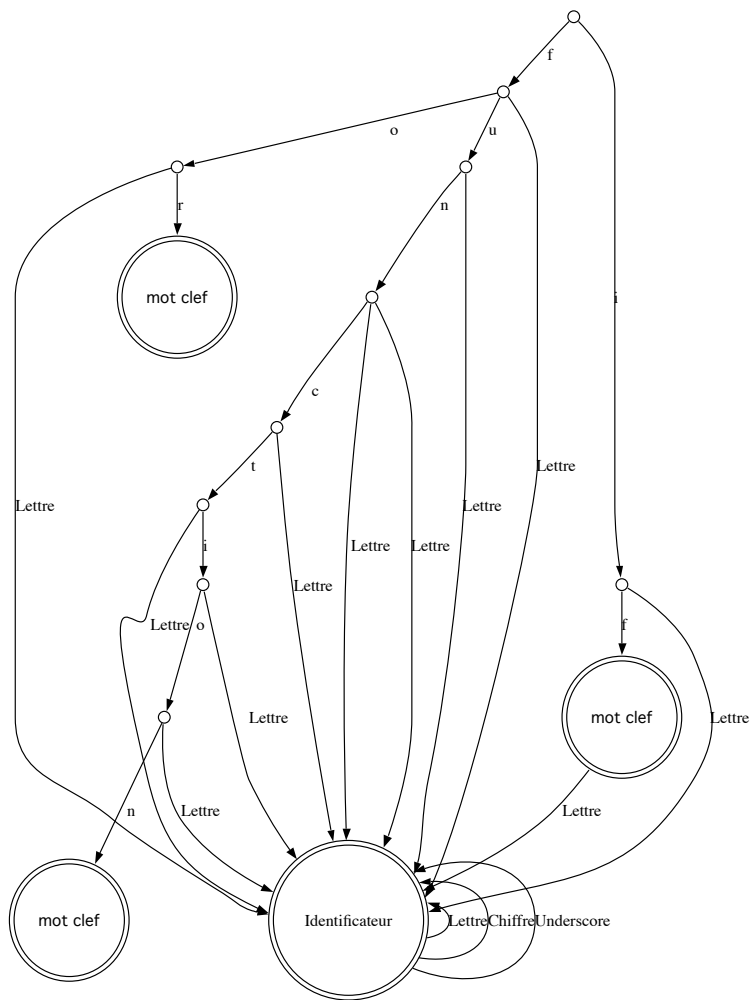


FIGURE 3 – Finite State Automata

```

%%

/* 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 */}

```

### 3 Analyse syntaxique

Quelques rappels

- Méthodes descendante
- Méthodes ascendantes
- Méthodes tabulaires

exemple avec ETF,  $a + a * (a + a)$

$E \rightarrow E + T$   
 $E \rightarrow T$   
 $T \rightarrow T * F$   
 $T \rightarrow F$   
 $F \rightarrow ( E )$   
 $F \rightarrow \text{IDENTIFIER}$

$a + b * (c + d)$   
 $F + b * (c + d)$   
 $T + b * (c + d)$   
 $E + b * (c + d)$   
 $E + F * (c + d)$   
 $E + T * (c + d)$   
 $E + T * (F + d)$   
 $E + T * (T + d)$   
 $E + T * (E + d)$   
 $E + T * (E + F)$   
 $E + T * (E + T)$   
 $E + T * (E)$   
 $E + T * F$

$$E + T$$

$$E$$

Dessiner arbre de dérivation

— Principe de l'analyse LR

— LR(0)

On ne regarde que la pile pour prendre une décision décalage/réduction (Grammar 3.20)

$$S' \rightarrow S \#$$

$$S \rightarrow ( L )$$

$$S \rightarrow x$$

$$L \rightarrow S$$

$$L \rightarrow L , S$$

$$\mathbf{Q1} S' \rightarrow \bullet S \#$$

$$S \rightarrow \bullet ( L )$$

$$S \rightarrow \bullet x$$

$$\mathbf{Q2} S \rightarrow x \bullet$$

$$\mathbf{Q3} S \rightarrow ( \bullet L )$$

$$L \rightarrow \bullet S$$

$$L \rightarrow \bullet L , S$$

$$S \rightarrow \bullet ( L )$$

$$S \rightarrow \bullet x$$

$$\mathbf{Q4} S' \rightarrow S \bullet \#$$

$$\mathbf{Q5} S \rightarrow ( L \bullet )$$

$$L \rightarrow L \bullet , S$$

$$\mathbf{Q6} S \rightarrow ( L ) \bullet$$

$$\mathbf{Q7} L \rightarrow S \bullet$$

$$\mathbf{Q8} S \rightarrow \bullet ( L )$$

$$S \rightarrow \bullet x$$

$$L \rightarrow L , \bullet S$$

$$\mathbf{Q9} L \rightarrow L , S \bullet$$

— SLR

la grammaire LR(0) de (3.23)

$S \rightarrow E \#$   
 $E \rightarrow T + E$   
 $E \rightarrow T$   
 $T \rightarrow x$

pose un problème avec " + " : on ne sait s'il faut réduire  
 $E \rightarrow T \bullet$

ou décaler  
 $E \rightarrow T \bullet + E$

Solution : on ne fait des réductions que sur les suivants des termes

- LR(1)  
 On a l'item  $(A \rightarrow \alpha \bullet \beta, x)$   
 $\alpha$  : sommet de la pile  
 Input  $\omega \mu : \beta x \Rightarrow \omega$
- LALR(1)
- Problème d'ambiguïté  
 Revenons à la grammaire ifthenelse

$\text{instr} \rightarrow \text{si expr alors instr}$   
 $\text{instr} \rightarrow \text{si expr alors instr sinon instr}$   
 $\text{instr} \rightarrow \text{autre}$   
 analyse de si E1 alors si E2 alors S2 sinon S3  
 Deux analyses :

1. si E1 alors (si E2 alors S2 sinon S3)
2. si E1 alors (si E2 alors S2) sinon S3

Grammaire équivalente :

$\text{instr} \rightarrow M$   
 $\text{instr} \rightarrow U$   
 $M \rightarrow \text{si expr alors } M \text{ sinon } M$   
 $M \rightarrow \text{autre } U \rightarrow \text{si expr alors instr}$   
 $U \rightarrow \text{si expr alors } M \text{ sinon } U$

- Grammaire ambiguë

$E \rightarrow \text{id}$   
 $E \rightarrow \text{num}$   
 $E \rightarrow E * E$   
 $E \rightarrow E / E$   
 $E \rightarrow E + E$   
 $E \rightarrow E - E$   
 $E \rightarrow ( E )$



Conflits décalage/réduction sur  $\{+, -, *, /\}$

$E * E + E$  deux arbres :

1.  $E(E * E(E + E))$  décalage
2.  $E(E(E * E) + E)$  réduction

Conflit entre

$E \rightarrow E * E \bullet$

$E \rightarrow E \bullet + E$

Pile :  $E * E$

$*$  prioritaire devant  $+$  : réduction plutôt que décalage

Autre type de conflit :

$E \rightarrow E + E \bullet$

$E \rightarrow E \bullet + E$

Le décalage correspond à un opérateur associatif à droite, la réduction à un opérateur associatif à gauche

$a + a + a = a + (a + a)$

Solution :

`precedence left PLUS, MINUS;`

`precedence left TIMES, DIVIDE, MOD;`

`precedence left UMINUS;`

- Correction d'erreurs

Quand une erreur de syntaxe est détectée, le parseur tente de remplacer la tête de l'input par *error* et continue de parser.

Exemple :

`stmt ::= expr SEMI |`

`while_stmt SEMI |`

`if_stmt SEMI |`

`error SEMI;`

## 4 Arbres de syntaxe abstraite

- Définition

Les noeuds de l'arbre : entités syntaxiques et sémantique Chaque nœud est tel que l'arbre qui a pour racine ce nœud est une composante du programme

Les feuilles correspondent aux entités lexicales

- La grammaire donne une syntaxe abstraite que l'on n'exploite pas immédiatement, on préfère calculer un arbre abstrait.
- Notions de grammaire attribuée
  - chaque production  $A \rightarrow \alpha$  possède un ensemble de règles  $b = f(c_1, c_2, \dots, c_k)$ 
    - $f$  fonction
    - $b$  attribut synthétisé de  $A$

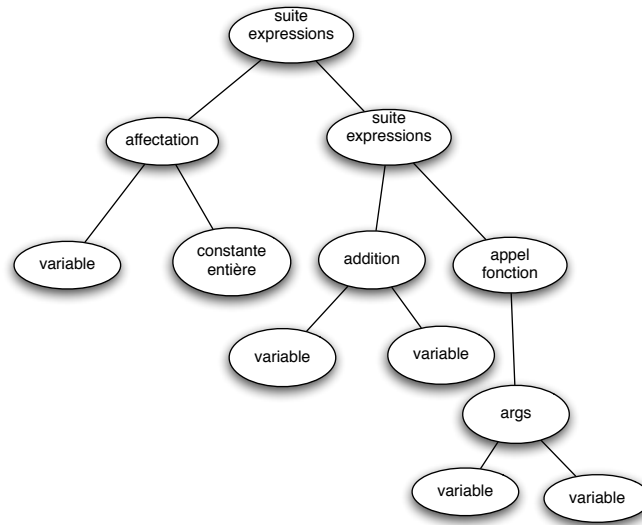


FIGURE 4 – Arbre de syntaxe abstraite

- ou alors  $b$  attribut hérité d'un des symboles en partie droite de la production
- $c_1, c_2, \dots, c_k$  attributs de symboles quelconques de la production

On parle de grammaire attribuée si  $f$  n'ont pas d'effet. Sinon on parle de définition dirigée par la syntaxe

exemple :

$L \rightarrow E$	$\text{imprimer}(E.val)$
$E \rightarrow E + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T * F$	$E.val = E_1.val \times T.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow ( E )$	$F.val = E.val$
$F \rightarrow \text{IDENTIFIER}$	$F.val = \text{IDENTIFIER}.vallex$

## 5 Types, vérification de type

Contrôles sémantiques statiques :

- Contrôle de type
  - Incompatibilité opérande/opérateur ou fonction
- Contrôle du flot d'exécution
  - Cohérence du flot : pas de `break` hors des boucles et `switch`, pas de `goto` sans étiquette valide
- Contrôle d'unicité
  - Présence unique d'objets dans certaines situations : variable définie une seule fois, `switch` non ambigu
- Contrôle de complétude et de cohérence
  - Présence de la déclaration d'un argument dans le contexte où il est utilisé. Utilisation d'un argument dans le contexte de sa déclaration.

## 5.1 Système de typage

1. Un type de base est une TypeExpr
2. Le nom d'un type est une TypeExpr
3. Un constructeur de type est une TypeExpr
  - (a) Tableau
  - (b) Produit
  - (c) Structures
  - (d) Pointeurs
  - (e) Fonctions
4. Une variable de type est une TypeExpr

Langage simple

Contrôle de type des expressions

Contrôle de type des instructions

Contrôle de type des fonctions

## 5.2 Equivalence structurelle des TypeExpr

Algorithme simple

— Environnement

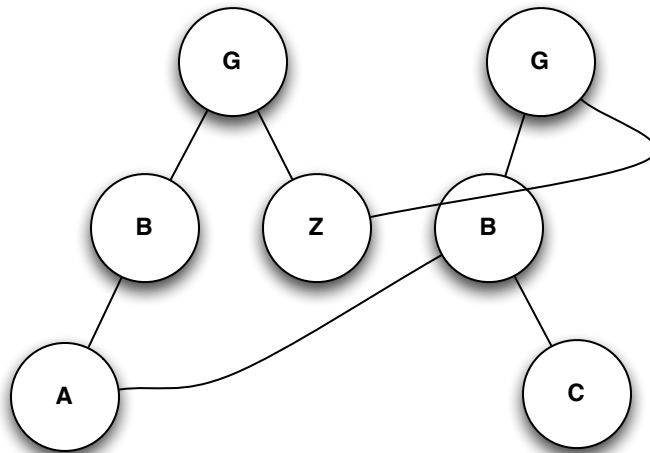


FIGURE 5 –

```
fonction ajouter(x:nom, t:type, u:Arbre):Arbre
debut
  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

```

— Représentation des types

1. Type de base (booléen, caractère, entier, réel), *void*, *error*
2. Nom de type
3. Constructeur :
  - (a) Tableaux (I, T)
  - (b) Produits
  - (c) Structures
  - (d) Pointeurs
  - (e) Fonctions
4. Variables de type

Lors de la compilation : statique, lors de l'exécution : dynamique

langages fortement typé : programmes sans erreur de type.

Graphe de représentation des expressions de type

Par un arbre si pas types rékursifs

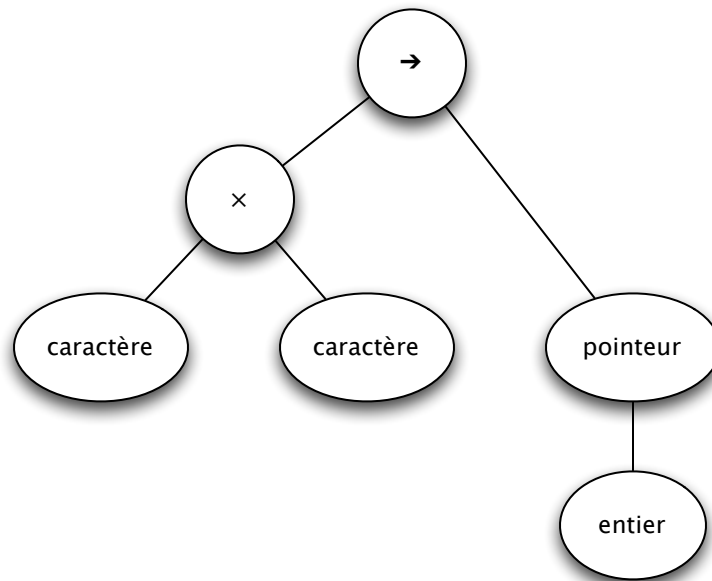


FIGURE 6 –

```

struct {
    int x;
    int y;
    int z [];
}

```

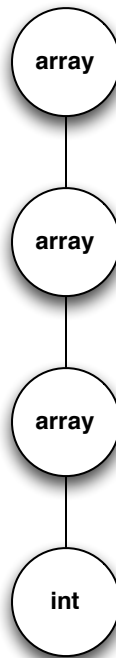


FIGURE 7 – `int a[ ][ ]`;

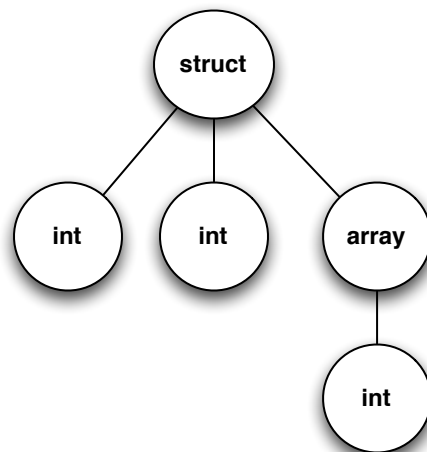


FIGURE 8 –

- Types du langage pseudo C
  - $P \rightarrow D; E$
  - $D \rightarrow D; D \mid \text{id} : T$
  - $T \rightarrow \text{char} \mid \text{int} \mid T[] \mid T^*$
  - $E \rightarrow \text{littéral} \mid \text{integer} \mid \text{identifier } E \bmod E \mid E[E] \mid *E$
- Vérification dans les expressions

### 5.3 Equivalence des expressions de type

Dans quelle mesure peut-on dire que deux types sont équivalents ?

**Solution simple :**

```

fonction Equiv(s, t): booléen
debut
  si s et t sont le meme 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 × s2 et t = t1 × t2 alors
    retourner Equiv(s1, s2) et Equiv(t1, t2)
  ...
  sinon
    retourner faux
fin
  
```

Problème : cycles

```

type lien = pointer of cellule;
type cellule = structure {
  info : int;
  suivant : lien;
}
  
```

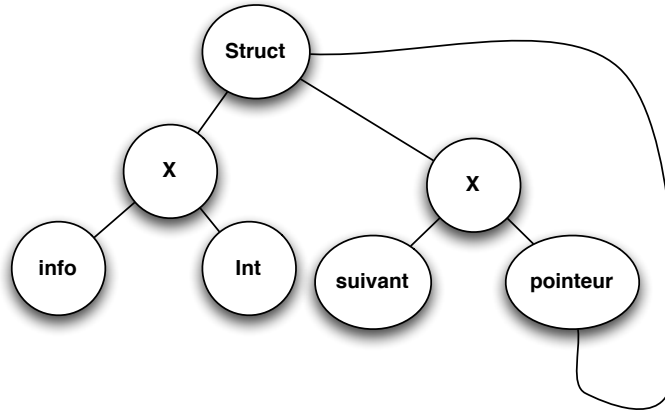


FIGURE 9 – Type cyclique pour une structure

## 5.4 Coercition, surcharge et polymorphisme

### Coercition

Exemple :

```

{
a : int;
b : real;
x : int;
y : real;
x = a + b;
y = a + b;
}

```

```

E → E :e1 op E :e2
{ : if e1.type == entier AND e2.type == entier
  RESULT.type = entier
  else if e1.type == reel OR e2.type == real
    RESULT.type = real
  else
    RESULT.type = error
  :}
Instr → LeftExpr :e1 = Expr :e2
{ : if e2.type == entier AND e1.type == real
  RESULT.code = Aff (RealToInt(e1.code), e2.code))
  else if e2.type == real AND e1.type == int
    RESULT.code = Aff (IntToReal(e1.code), e2.code))
  :}

```

### Surcharge

surcharge : signification différente suivant contexte. exemple :

Addition : addition d'entiers, de réels, de complexes, concaténation de chaînes.

Problème :

```
function foo (i, j : int) : real;
function foo (i, j : int) : int;

  int × int → int
  int × int → real

foo(4, 5) ???
i : int;
x : real;
foo(4, 5) + x;
foo(4, 5) + i;

E → ID :id { : RESULT.types = rechercher (id) :}
E → E :e1 (E :e2) { : RESULT.types = {t | ∃s, s ∈ e2.types  s → t ∈ e1.types} :}
```

## Polymorphisme

Tout morceau de code que l'on peut exécuter avec des arguments de types différents.

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

```
# let rec long = function
  [] -> 0
  | t :: q -> 1 + long q;;
val long : 'a list -> int = <fun>
```

Exemple de polymorphisme :pp. 406 407

```
P → D; E
D → D; D
  | id : Q
Q → ∀ id . Q
  | T
T → T → T
  | T × T
  | pointeur(T)
  | liste(T)
  | type
  | ID
  | ( T )
E → E ( E )
  | E, E
  | ID
```



```

deref :  $\forall x . \text{pointeur}(x) \rightarrow x$ 
q :  $\text{pointeur}(\text{pointeur}(\text{entier}))$ 
deref (deref (q))

```

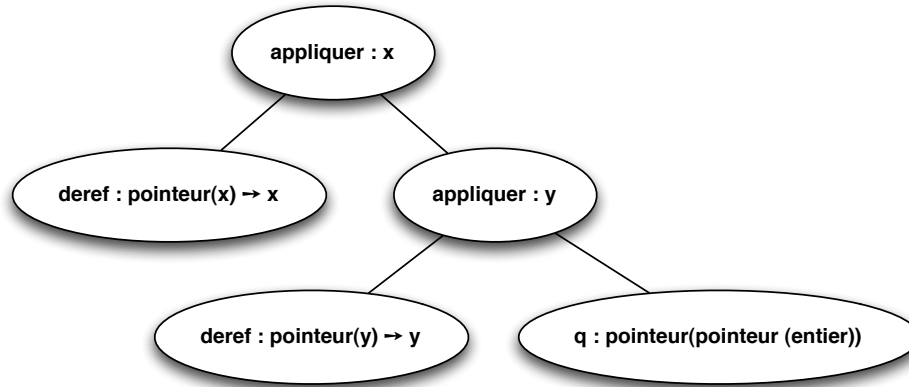


FIGURE 10 –

$\text{pointeur}(y) = \text{pointeur}(\text{pointeur}(\text{entier}))$      $\text{ppcu} = \{ \langle y, \text{pointeur}(\text{entier}) \rangle \}$      $\text{pointeur}(x) = \text{pointeur}(\text{entier})$      $\text{ppcu} = \{ \langle x, \text{entier} \rangle \}$

## 6 Génération de code intermédiaire

### 6.1 Code à trois adresses

- $x := y \text{ op } z$
- $x := \text{op } y$
- $x := y$
- $\text{jumpL}$
- $\text{if } x \text{ op } y \text{ jumpL}$
- $\text{param } x, \text{ call } p, n, \text{ return } y$
- $x := y[i], x[i] := y$
- $x := \&y, x := *y, *x := y$

Traduction dirigée par la syntaxe pour du code à 3 adresses (p. 517)

### 6.2 Déclarations

Entré dans la table des symboles avec type + adresse relative à l'emplacement

- Dans une procédure
- Dans une structure
- Dans un tableau
- Instructions de flot de contrôle
- Dans un tableau
- Appel de procédures

## 7 Organisation de l'espace de travail

Soit le bloc de mémoire du code compilé :

1. Code cible
2. Données Statiques
3. Pile
4. Tas

### 7.1 enregistrement d'activation

Appel d'une fonction : un espace est alloué (dans la pile) (p.439)

1. adresse de retour
2. paramètres
3. état machine
4. données locales
5. temporaires

adresse de retour :  $a + 1$  où  $a$  est l'adresse de l'appel

### 7.2 Allocation

#### 7.2.1 Allocation statique

L'adresse consiste à décaler

- Taille connue à la compilation - Pas de récursivité! - Pas d'allocation dynamique

#### 7.2.2 Allocation en pile

- Pas de perte (variables locales supprimées) - Taille connue

exemple : protocole d'appel des procédures

#### Appel

1. Évalue les arguments
2. Stocke adresse retour
3. Sauvegarde registres et état courant
4. Initialise données locales

#### 7.2.3 Allocation dans le tas

### 7.3 passage paramètres

1. Valeur  
Valuer dans enregistrement d'activation
2. Référence  
pointeur

3. Copie-restauration (valeur-résultat)
4. Nom

## 8 Du code intermédiaire vers le code optimisé

### 8.1 Arbres canoniques

ESEQ, CALL → ordre pertinent

1. Un arbre est réécrit sous la forme d'une liste d'arbres canoniques sans ESEQ ni SEQ
2. Chaque liste est regroupée dans un bloc basic qui ne contient ni saut ni label
3. CJUMP → immédiatement suivi par partie fausse du test
  1. Pas de SEQ ou de ESEQ
  2. CALL dans EXP(...) ou MOVE(TEMP t, ...)

## 9 Graphe du flot de contrôle

### 10 Variables d'un bloc

exemple :

B1:

a = 0

B2:

LABEL 11

b = a+1

c=c+b

a=a\*2

if a < n goto 11 else goto 12

B3:

LABEL 12

print c

B1 → B2

B2 → B2

B2 → B3

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

$Out(B)$  : vivantes en sortie de B

$In(B)$  : vivantes en entrée de B

$Use(B)$  = figure dans un membre droit dans B avant de figurer dans un membre gauche ou d'être défini

$Def(B)$  = figure dans un membre gauche dans B

$Out(B) = \cup_{B' \text{ successeurs de } B} In(B')$

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

Algo :

1. Déterminer Use et Def
2. Initialiser  $In$  et  $Out$  à  $\emptyset$
3. L1 : Appliquer les équations
4. si modification, retourner en L1

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, t3)
```