Traduction des langages

Aurélie Hurault
hurault@enseeiht.fr

Plan

I. Introduction

II. Analyse lexicale : Rappels et compléments

Plan

III. Analyse syntaxique : Les grammaires

IV. Arbre Syntaxique Abstrait (AST)

Plan

V. Analyse sémantique : Application à la compilation

VI. Conclusion

Première partie I

Introduction

Introduction

Objectif du cours

- Comprendre le fonctionnement d'un compilateur
- Etre capable d'en écrire un

Qu'est ce qu'un compilateur ?

Plan

Interpréteur versus compilateur

Machine virtuelle

Une machine qui reconnaît un certain nombre d'instructions qui ne sont pas (toutes) "natives" pour la machine hardware.

Interpréteur

Un programme qui prend en entrée un autre programme, écrit pour une machine virtuelle, le traduit et l'exécute.

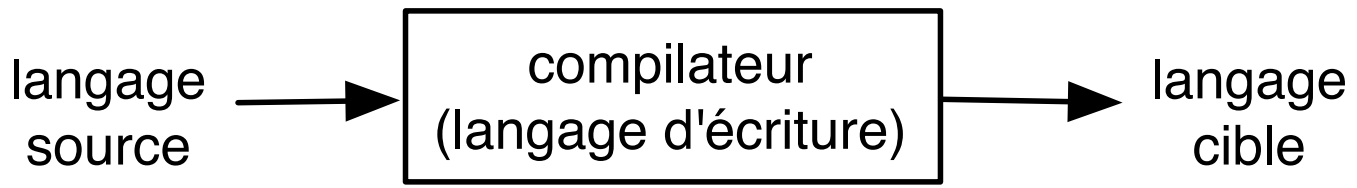
Compilateur

Un programme, qui traduit un programme écrit dans un langage L dans un programme écrit dans un langage L' différent de L (en général L est un langage évolué et L' est un langage de plus bas niveau).

Compilateur

Remarque

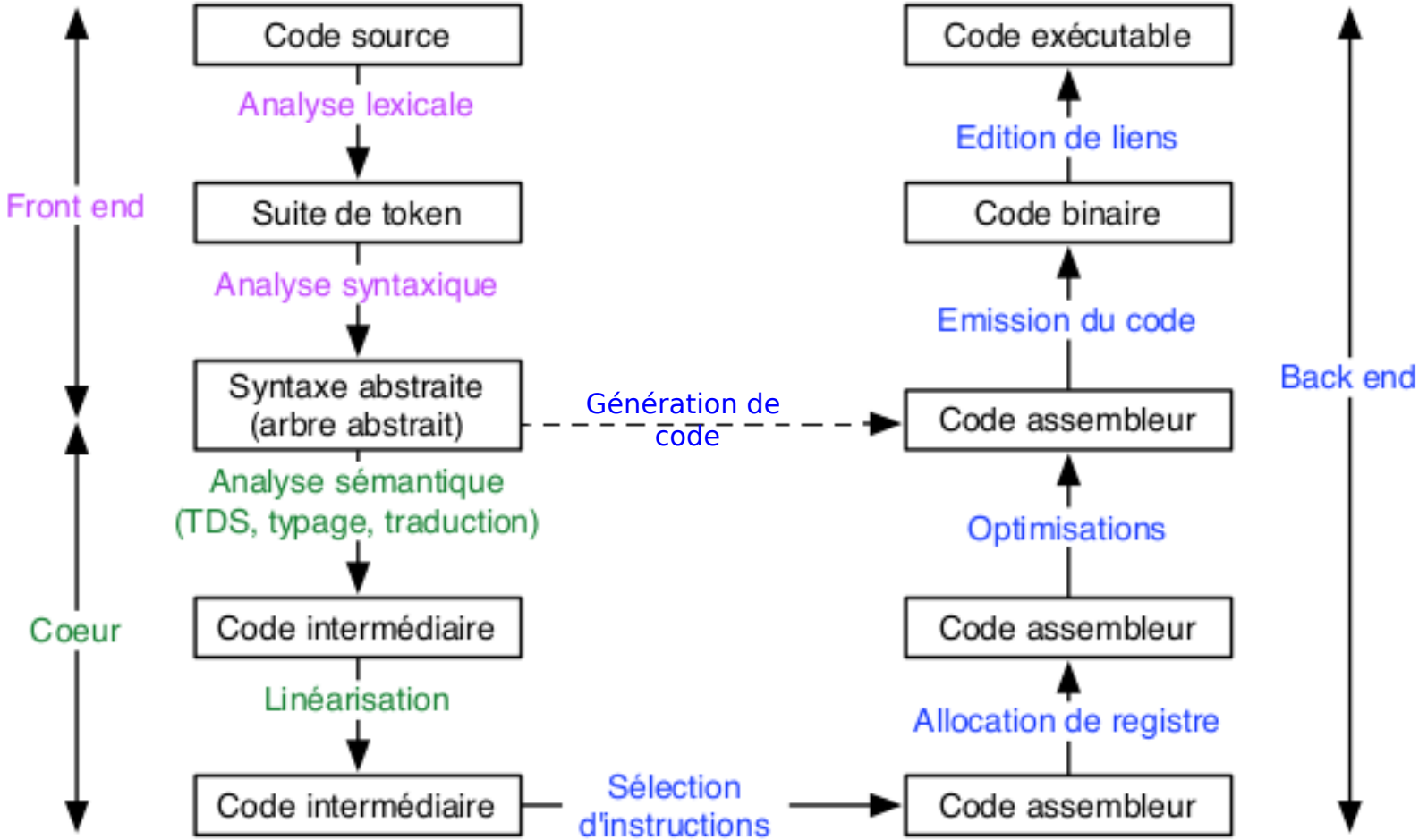
Trois langages : langage source, langage cible et langage d'écriture du compilateur.



Le langage cible peut être :

- du langage assembleur / machine
  - \+ efficace
  - \- pas portable
- un langage évolué
  - \+ portable
  - \- pas efficace

Du source à l'exécutable



Exemple(1)

- Code source :

```
int x = 3;
int y = x+1;
print y;
```

- Code cible (CRAPS) :

```
setq 3, %r1
mov %r1, %r2
inccc %r2
mov %r2, %r0
call print
stop: ba stop
```

- Affichage du contenu de r0

```
SSG = 0xA0000000
print: set
SSG, %r3
setq 0b1111, %r4
st %r4, [%r3+1]
st %r0, [%r3]
ret
```

Front-end

- Lié au langage source
- Analyse lexicale : découpe le code source en suite de terminaux.
- Analyse syntaxique : s'assurer que l'on sait traiter le code source
  - Le code source doit respecter une syntaxe particulière
  - Cette syntaxe est donnée à l'aide d'une grammaire

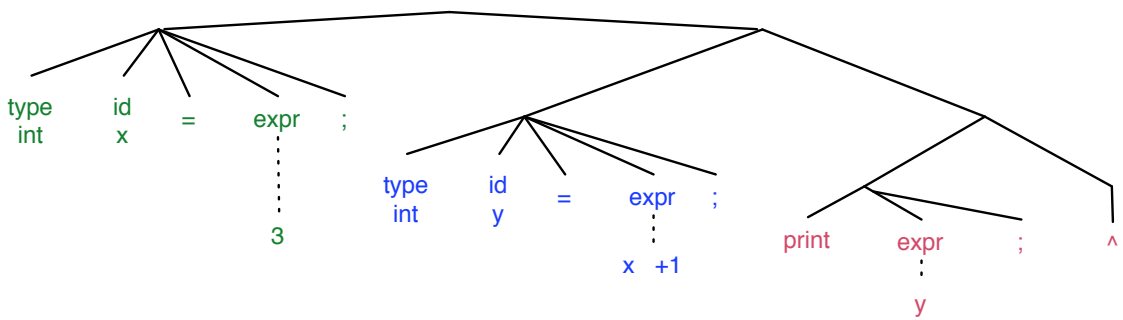
Exemple(2)

- Reconnaître les différents composants (appelés terminaux) du programme
  - `int`, `=`, `;`, `+`, `print` : mots clés du langage
  - `x`, `y` : identifiants
  - `3`, `1` : valeurs numériques

int | x | = | 3 | ; | int | y | = | x | + | 1 | ; | print | y | ;

⇒ Analyse lexicale

- Respect d'une syntaxe particulière : représentation du programme sous forme d'arbre de dérivation



⇒ Analyse syntaxique

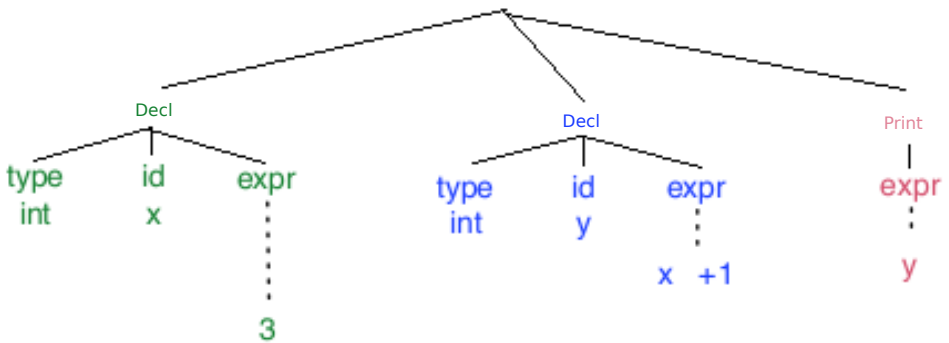
Coeur : Analyse sémantique

- La phase "sémantique" pourrait être toute la traduction de la syntaxe abstraite vers le code machine : un sens est donné au langage à l'aide des moyens d'expression de la machine.
- Pour des raisons d'optimisation, on peut produire un code intermédiaire (factorisation du back end).

- La suite de ce cours sera particulièrement dédiée à l'analyse sémantique. Certaines étapes ne seront pas détaillées :
  - pas de code intermédiaire et donc pas de linéarisation ;
  - pas d'optimisation ;
  - le code assembleur généré sera interprété par une machine dédiée.

Exemple(3)

- "Traiter" le code source
  - Utilisation de la représentation du programme sous forme d'arbre syntaxique abstrait (AST)



  - On remarque que les deux x sont liés : circulation d'informations dans l'arbre
    - parcours de l'AST

$\Rightarrow$ Analyse sémantique

Exemple(4)

- Informations à vérifier à la compilation :
  - bonne utilisation des variables
  - typage
  - ...

  $\Rightarrow$ besoins d'informations (nom, type, ...) sur les identificateurs : table des symboles

Exemple(5)

- Exemple de code intermédiaire :

```
Seq[
    Move_Mem(3,x),
    Move_Mem(Call(Plus1,Acces_Mem(x)),y),
    Call(Print,Acces_Mem(y))
    ];
```

Coeur : Linéarisation

Un aspect qui est commun à toutes les machines connues est que le code est une liste d'instructions. Or, le code intermédiaire peut avoir une structure arborescente. Le but de la phase de linéarisation est de mettre le code à plat.
Nous risquons pour cela d'avoir besoin de variables intermédiaires ou temporaires.

Exemple(6)

- Code intermédiaire après linéarisation :

```
Move_Mem(3,x);
Move_Temp(Acces_Mem(x),t1);
Move_Temp(Call(Plus1,t1),t2);
Move_Mem(t2,y);
Move_Temp(Acces_Mem(y),t3);
Call(Print,t3);
```

Back-end (1)

- Lié à la machine cible
- Sélection d'instructions
  - Passage du code intermédiaire au code assembleur de la machine ciblée.
  - La sélection opérée sur une instruction du code intermédiaire consiste à parcourir l'arbre de cette instruction en émettant la ou les instructions machines qui la réalisent.
  - Si plusieurs séquences d'instructions possibles, il faut faire un choix.

Exemple(7)

- Exemple de choix :
  - `Call(Plus1,...)` : appel d'une méthode d'incrémentation ou d'une addition ?
  - `Move_Mem(3,x);` : appel de `set` ou `setq` ?
- x dans r1, y dans r2 et les temporaires dans r3 et plus...
- Code assembleur après sélection d'instruction (CRAPS) :

```
setq 3, %r1
mov %r1, %r3 //t1 dans r3
inccc %r3
mov %r3, %r4 //t2 dans r4
mov %r4, %r2
mov %r2, %r5 //t3 dans r5
mov %r5, %r0
call print
stop: ba stop
```

Back-end (2)

- Allocation de registre
  - Une première phase d'analyse de durée de vie sert à déterminer les information de durée de vie des temporaires.
  - La mission de l'allocation de registres est alors de transformer les temporaires arbitraires du code assembleur produit par la sélection en registres de la machine ciblée.

Exemple(8)

- Avec l'analyse de durée de vie des temporaires, on voit qu'aucun ne chevauche les autres, on peut donc utiliser un unique registre : r3
- Code assembleur après allocation de registre :

```
setq 3, %r1
mov %r1, %r3;
inccc %r3
mov %r3, %r3;
mov %r3, %r2
mov %r2, %r3
mov %r3, %r0
call print
stop: ba stop
```

Back-end(3)

- Optimisations
  - Suppression de code mort.
  - Optimisation des boucles.
  - Factorisation.
  - ...
- Emission de code
  - Le code peut alors être émis dans un fichier.

Exemple(9)

- Code assembleur après optimisation :

```
setq 3, %r1
mov %r1, %r2
inccc %r2
mov %r2, %r0
call print
stop: ba stop
```

Back-end(4)

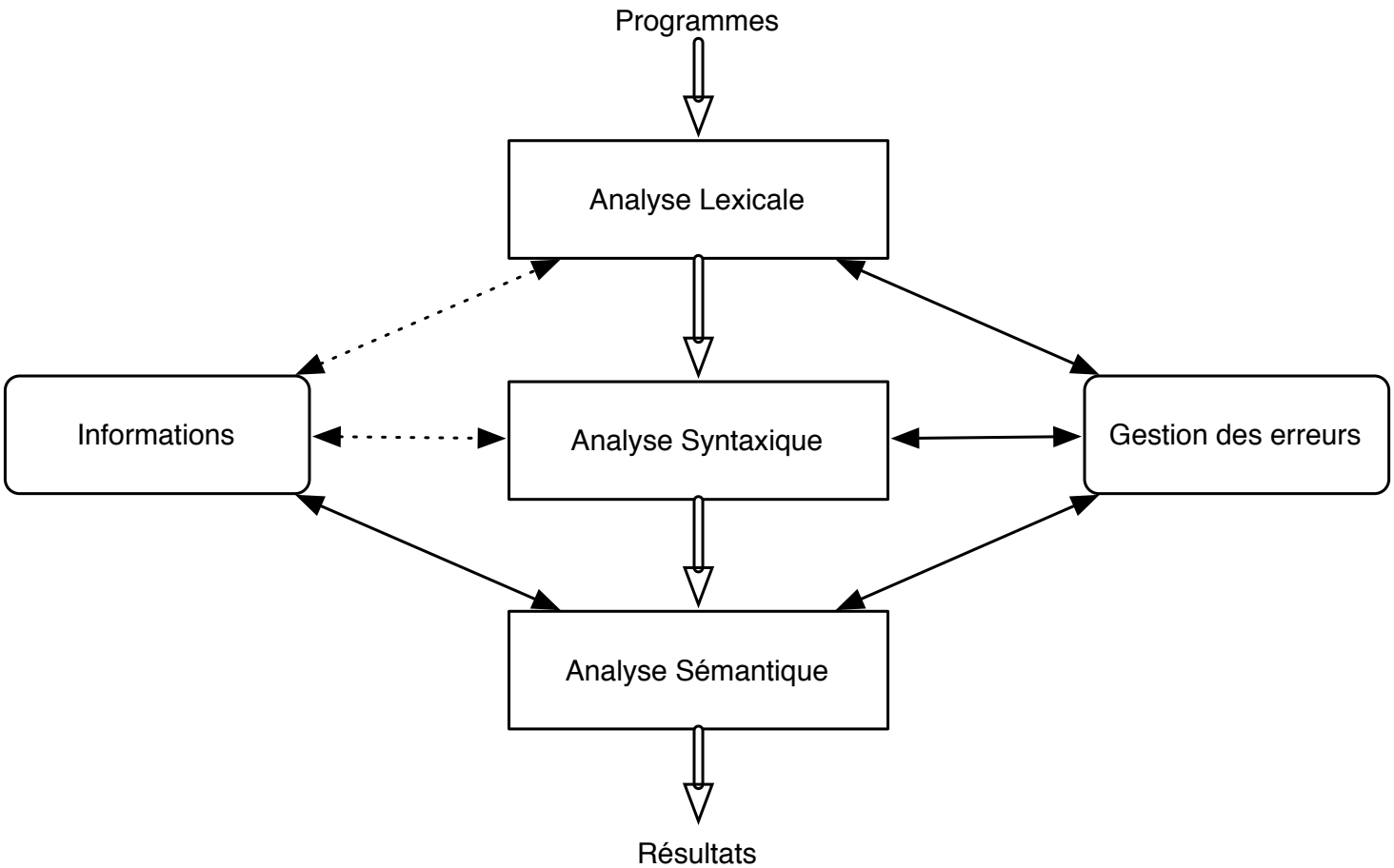
- Edition de liens
  - Quand un programme est réparti sur plusieurs fichiers et qu'il y a des références entre eux, il faut pouvoir accéder aux informations des différents fichiers.
  - C'est un autre programme, dit éditeur de liens qui prend tous les fichiers émis et fabrique l'exécutable. L'éditeur de lien s'occupe essentiellement de mettre tous les fichiers émis les uns derrière les autres et de résoudre les références symboliques entre ces fichiers.

Plan

Les points abordés

- Analyse lexicale
  - Retour sur alphabet, mot, langage, automate fini à états, expression régulière (cf cours de première année)
- Analyse syntaxique
  - Grammaire générale
  - Grammaire algébrique (ambiguïté, transformation, ...)
  - Analyse Descendante Récursive
- AST
- Analyse sémantique et Concepts à traiter
  - Table de symboles
  - Typage
  - Gestion de la mémoire
  - Génération de code

Synthèse


Programmes
Analyse Lexicale
Informations
Analyse Syntaxique
Gestion des erreurs
Analyse Sémantique
Résultats

Deuxième partie II

Analyse lexicale : Rappels et compléments

Plan

Automate fini (déterministe) à état

Automate fini (déterministe) à état

Un automate (fini déterministe) est un quintuplet $A = (Q, X, \delta, q_I, F)$ où :

- Q : ensemble fini d'états
- X : alphabet
- $q_I \in Q$: l'état initial de l'automate
- $F \subseteq Q$: les états finals (ou terminaux)
- $\delta \in Q \times X \mapsto Q$: fonction de transition de l'automate.

Automate et langage

Extension de δ

$$\begin{cases} \hat{\delta}(q, \Lambda) & = & q \\ \hat{\delta}(q, au) & = & \hat{\delta}(\delta(q, a), u), \ a \in X, \ u \in X^* \end{cases}$$

Configuration

Une configuration est un couple (q, m) avec $q \in Q$ et $m \in X^*$

Transitions

Relation $\vdash$ entre configurations : $(q, am) \vdash (q', m)$ si $q' = \delta(q, a)$.
$\vdash^*$ fermeture réflexive transitive de $\vdash$

Langage accepté

$$\begin{aligned} L(A) &= \{m \in X^* \mid \hat{\delta}(q_I, m) \in F\} \\ &= \{m \in X^* \mid \exists q_F \in F : (q_I, m) \vdash^* (q_F, \Lambda)\} \end{aligned}$$

Propriétés des langages rationnels

Langage rationnel

L est rationnel $\equiv \exists A$ automate : $L = L(A)$
Rat = ensemble des langages rationnels (ou réguliers)

Fermeture

Rat est fermé par union, produit, étoile, intersection, complémentaire, différence et miroir.
Soient $L_1, L_2 \in$ Rat, alors :

$$L_1 \cup L_2 \in \text{Rat} \qquad L_1 \bullet L_2 \in \text{Rat}$$
$$L_1{}^* \in \text{Rat} \qquad L_1 \cap L_2 \in \text{Rat}$$
$$\overline{L_1} \in \text{Rat} \qquad L_1 \setminus L_2 \in \text{Rat}$$

Expressions Régulières

Soit X un alphabet fini, et $Y = \{(,), ^*, +, \bullet, \Lambda, \emptyset\}$ un alphabet disjoint. Un mot m de $(X \cup Y)^*$ est une expression régulière sur X ssi :

- soit m est $\emptyset$ ou Λ ou un symbole de X,
- soit m est de la forme $(x + y)$ ou $(x \bullet y)$ ou x^*, où x et y sont des expressions régulières sur X.

Langage associé

Une expression régulière m sur X définit un langage $L(m)$ sur X d'après les règles suivantes :

- $L(\emptyset)$ est le langage vide ;
- $L(\Lambda) = \{\Lambda\}$;
- Si $a \in X$, alors $L(a) = \{a\}$;
- Pour tout expression régulière u et v sur X,

$$\begin{aligned} L(u+v) &= L(u) \cup L(v) \\ L(u \bullet v) &= L(u)L(v) \\ L(u^*) &= L(u)^* \end{aligned}$$

Equivalence

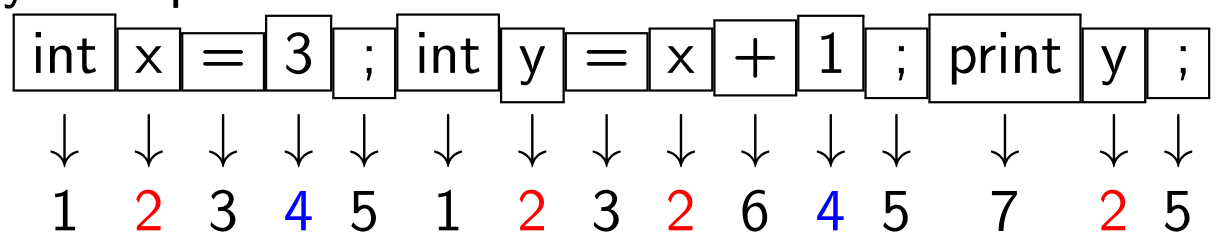
Il y a équivalence entre :

- les langages rationnels (Rat) ;
- les langages reconnus par les AFN ;
- les langages reconnus par les AFD ;
- les langages définis par les expressions régulières.

Plan

Analyse lexicale

- Découpe un flux continu en terminaux (ou unité lexicale : code qui correspond à un symbole)
- Les terminaux sont exprimés par une expression régulière
- L'analyse pourrait se faire caractère par caractère, cependant à terme nous voulons reconnaître des phrases et donc des suites de mots, pour être plus efficace l'analyse lexicale reconnaît des mots.
- Le résultat de l'analyseur lexical est donné à l'analyseur syntaxique $\Rightarrow$ associer un code aux terminaux.



Analyse lexicale : outil

- Monde C : Lex
- Monde Java : JFlex
- Monde OCaml : Ocamllex
- ...

Analyse lexicale : outil JFLEX

- Outil d'analyse lexicale dans le monde Java.
- Format des fichiers

```
code utilisateur
%%
options et déclarations
%%
règles lexicales
```

- Code utilisateur :
  - Le code à mettre en entête du fichier généré.
- Options et déclarations :
  - Définition de macro :
    `identificateur = expression régulière`
- Règles lexicales :
  - Associe une action à une expression régulière.

Analyse lexicale : un exemple avec JFLEX

```
/* Code qui sera ajouté tel quel au début de la classe engendrée.
 * Utile pour les directives  package et import
 */
package analyseur;
%%    // Commence la partie "Options et declarations"
/* Nom de la classe engendrée par JFLEX */
%class Lexical
/* Rend disponible les numéros de ligne et colonne */
%line
%column
%int  //Les actions sont de type de retour int.
%scanerror AnalyseException
/* Mode autonome : une méthode principale est définie dans la classe */
%standalone
/* Le code entre %{ %} est inclus tel quel dans la classe engendrée */
%{
   final static int CODE_INT = 1;
   final static int CODE_PRINT = 2;
   final static int CODE_EQUALS = 3;
   final static int CODE_PLUS = 4;
```

Analyse lexicale : un exemple avec JFLEX

```
   final static int CODE_SEMICOLON = 5;
   final static int CODE_NATURAL = 6;
   final static int CODE_ID = 7;
   public int lineno(){return yyline+1;}
   public int column() {return yycolumn+1;}
%}
/* Définition de macros */
...

%%    // Commence la partie "Régles et actions"
[ \n\r\t\f]             { /* Ignoré */ }
"--"[^\n]*\n            { /* Ignoré */ }
"//"[^\n]*\n            { /* Ignoré */ }
int                     { return CODE_INT; }
print                   { return CODE_PRINT; }
"="                     { return CODE_EQUALS; }
"+"                     { return CODE_PLUS; }
";"                     { return CODE_SEMICOLON; }
[0-9]+                  { return CODE_NATURAL; }
[a-zA-Z][a-zA-Z0-9]*    { return CODE_ID; }
.         { throw new AnalyseException("Lexical error : line "+yyline+" :
           invalid character : <" + yytext() + ">" ); }
```

Analyse lexicale : un exemple avec JFLEX

Utilisation de l'analyseur lexical généré.

```
...
try {
  Lexical  scanner = new Lexical( new java.io.FileReader("monFichier") );
  int nb;
  do{
    nb = scanner.yylex();
    System.out.println(scanner.yytext()+":"+nb);
    }while( nb != Lexical.YYEOF );
}
catch(...){...}
...
```

Plan

Bilan

On est maintenant capable de découper notre programme en terminaux, qui correspondent aux mots de notre langage.

Troisième partie III

Analyse syntaxique : Les grammaires

Plan

Grammaire formelle

Grammaire formelle

Une grammaire formelle (Chomsky 1956) est un quadruplet $G = (V, X, P, S)$ où :

- V est un ensemble fini nommé alphabet non terminal ;
- X est un ensemble fini disjoint de V nommé alphabet terminal ;
- $S \in V$ est l'axiome de départ ;
- P est un sous-ensemble fini de $(V \cup X)^+ \times (V \cup X)^*$. Un élément de P est une production et est noté $u \to v$.

Grammaire formelle

Exemple de grammaire formelle

$G = (V, X, P, S)$

- $V = \{A, B, C\}$
- $X = \{x, y, z\}$
- $S = A$
- $P = \{$
  - $A \rightarrow xyBCz$
  - $yB \rightarrow xA$
  - $AC \rightarrow z$

  $\}$

Dérivation

Dérivation

Soient x et y dans $(V \cup X)^*$. x se dérive en y pour la grammaire G (noté $x \Rightarrow y$) s'il existe z_1, z_2, u et v tels que $x = z_1 u z_2$ et $y = z_1 v z_2$ et $(u \to v) \in P$.

$\overset{*}{\Rightarrow}$

On note $\overset{*}{\Rightarrow}$ la fermeture transitive de $\Rightarrow$.

Langage engendré par G

$L(G) = \{m \in X^* \mid S \overset{*}{\Rightarrow} m\}$.

Exemple de dérivation

- Soit la grammaire G
  1. $S \rightarrow aSb$
  2. $S \rightarrow c$
- Dérivation : $S \Rightarrow_1 aSb \Rightarrow_1 aaSbb \Rightarrow_2 aacbb$
- Langage : $L(G) = \{a^n c b^n | n \geq 1\}$

Grammaire algébrique

Grammaire algébrique

Une grammaire formelle G est algébrique (ou context-free ou non-contextuelle) si chaque règle de production est de la forme : $A \to w$, avec $A \in V, w \in (V \cup X)^*$.

Langage algébrique

Un langage L est algébrique ou context-free s'il existe une grammaire algébrique qui l'engendre.

Alg

Alg = la famille des langages algébriques.

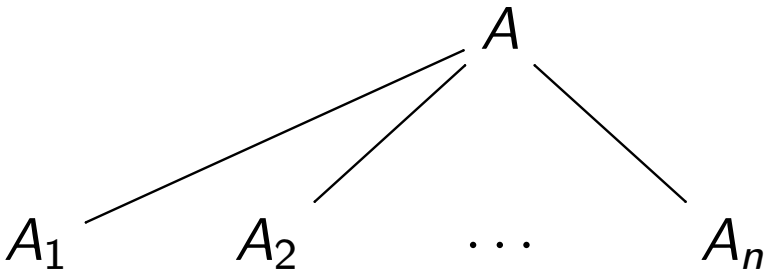
Exemple de grammaire algébrique

- La grammaire précédente (pour l'exemple de la dérivation).
- La grammaire des expressions :
  1. $E \rightarrow E + E$
  2. $E \rightarrow E * E$
  3. $E \rightarrow id$

Plan

Arbre de dérivation

Utilisation de $A \rightarrow A_1 A_2 .. A_n$:



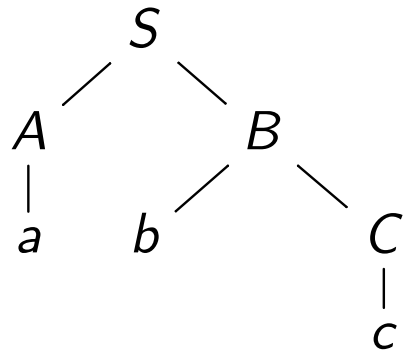
Soit G :
$$\begin{cases} 1. & S & \rightarrow AB \\ 2. & A & \rightarrow a \\ 3. & B & \rightarrow bC \\ 4. & C & \rightarrow c \end{cases}$$

$abc \in L(G)$:

$\underline{S} \overset{1}{\Rightarrow} \underline{A}B \overset{2}{\Rightarrow} a\underline{B} \overset{3}{\Rightarrow} ab\underline{C} \overset{4}{\Rightarrow} abc$

$\underline{S} \overset{1}{\Rightarrow} A\underline{B} \overset{3}{\Rightarrow} \underline{A}bC \overset{2}{\Rightarrow} ab\underline{C} \overset{4}{\Rightarrow} abc$

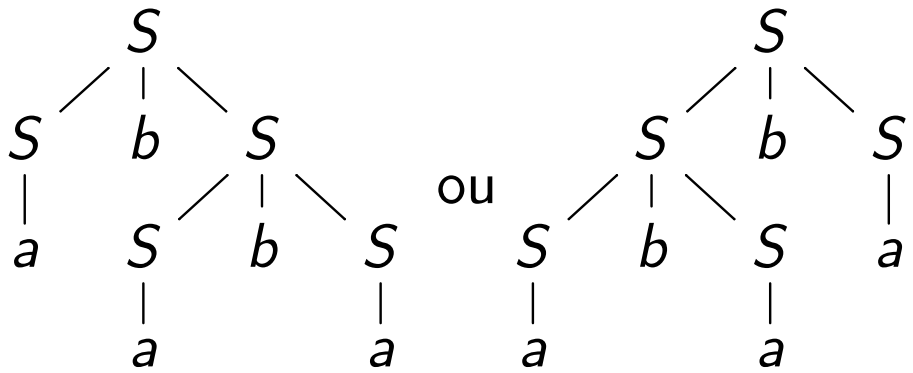
$\underline{S} \overset{1}{\Rightarrow} A\underline{B} \overset{3}{\Rightarrow} Ab\underline{C} \overset{4}{\Rightarrow} \underline{A}bc \overset{2}{\Rightarrow} abc$



Ambiguïté

Soit $G : \begin{cases} 1. & S \to SbS \\ 2. & S \to a \end{cases}$

$ababa \in L(G)$:



$G' : \begin{cases} 1. & S \to Ta \\ 2. & T \to abT \\ 3. & T \to \Lambda \end{cases}$

$L(G') = L(G) = \{(ab)^n a \mid n \geq 0\} = (ab)^* a$

Langage inhéremment ambigu

- Il existe des langages algébriques inhéremment ambigus, ie dont toutes les grammaires sont ambiguës.
- Exemple : $L = \{a^n b^p c^q \mid n = p \text{ ou } p = q\}$
  L est inhéremment ambigu.
- Intuitivement, il faut un procédé pour $n = p$, et un procédé pour $p = q$, d'où deux méthodes pour $n = p = q$.

Exercice

On considère la grammaire suivante :

- $A \rightarrow aAb$
- $A \rightarrow AA$
- $A \rightarrow bAa$
- $A \rightarrow \Lambda$

1. Donner des mots de $L(G)$ de taille 0, 2, 4.
2. Tous les mots commençant par a se terminent t'il par b ?
3. Intuitivement, quel est le langage engendré par G ?
4. Soit le mot $w = aabbbaab$. Donner pour w, une dérivation droite, une dérivation gauche et un arbre de dérivation.
5. G est elle ambiguë ?

Exercice

Pour chacun de langage ci-dessous, donner une grammaire algébrique qui l'engendre :

1. langage des palindromes sur $\{a, b\}$
2. $\{a^n b^p | n \geq p \geq 0\}$

Analyseur syntaxique

- Un analyseur syntaxique est un programme qui pour $m \in X^*$:
  - vérifie que $m \in L(G)$;
  - construit (implicitement ou explicitement) l'arbre de dérivation.
- L'analyseur syntaxique peut être écrit à la main, ou engendré automatiquement à partir de la grammaire.

Construction de l'arbre

- **Ascendant**
  - A partir des sources : reconnaître un morceau et le remplacer par son non-terminal.
  - Pas évident, car plusieurs choix possibles.
  - Pas intuitif, mais il existe des méthodes de construction sur ce principe (LR(k)).
- **Descendant**
  - A partir de l'axiome : dériver jusqu'à la source.
  - Plusieurs choix possibles : retour en arrière.
  - Pas efficace dans le cas général, mais plus simple qu'ascendant $\Rightarrow$ ajout de contrainte sur la grammaire pour que ça soit simple et efficace.

Construction de l'arbre : Exemple

1. $S \rightarrow A\ B$
2. $A \rightarrow a\ A$
3. $B \rightarrow b\ B$
4. $A \rightarrow a$
5. $B \rightarrow b$

But : reconnaître le mot "aaabb".

- **Ascendant**
  Lecture : $aaabb$
  $aaabb$

$a\ \ a\quad a\ \ b\quad b$

Construction de l'arbre : Exemple

1. $S \rightarrow A\ B$
2. $A \rightarrow a\ A$
3. $B \rightarrow b\ B$
4. $A \rightarrow a$
5. $B \rightarrow b$

But : reconnaître le mot "aaabb".

- **Ascendant**
  Lecture : *aa*abb
  aaabb

$a\ \ a\quad a\ \ b\quad b$

Construction de l'arbre : Exemple

1. $S \rightarrow A\ B$
2. $A \rightarrow a\ A$
3. $B \rightarrow b\ B$
4. $A \rightarrow a$
5. $B \rightarrow b$

But : reconnaître le mot "aaabb".

- **Ascendant**
  Lecture : *aaabb*
  aaabb

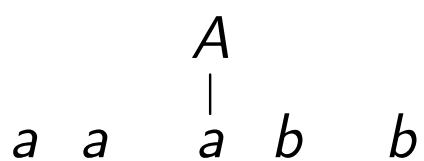
$a\ \ a\ \ \ a\ \ b\ \ \ b$

Construction de l'arbre : Exemple

1. $S \rightarrow A\ B$
2. $A \rightarrow a\ A$
3. $B \rightarrow b\ B$
4. $A \rightarrow a$
5. $B \rightarrow b$

But : reconnaître le mot "aaabb".

- **Ascendant**
  Lecture : *aaabb*
  $aaabb \overset{4}{\Leftarrow} aaAbb$

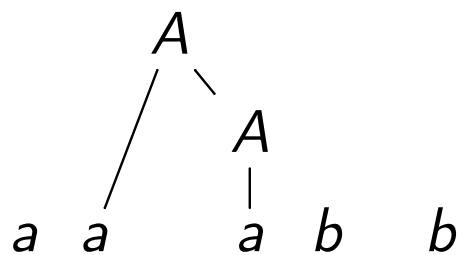


Construction de l'arbre : Exemple

1. $S \rightarrow A\ B$
2. $A \rightarrow a\ A$
3. $B \rightarrow b\ B$
4. $A \rightarrow a$
5. $B \rightarrow b$

But : reconnaître le mot "aaabb".

- **Ascendant**
  Lecture : $aaabb$
  $aaabb \overset{4}{\Leftarrow} aaAbb \overset{2}{\Leftarrow} aAbb$



Construction de l'arbre : Exemple

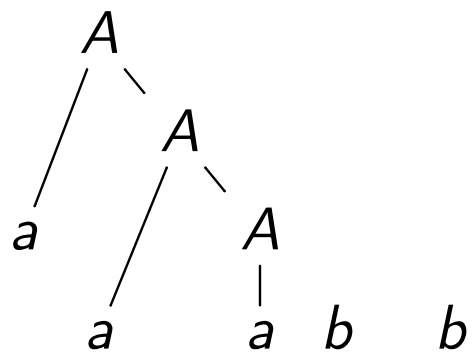
1. $S \to A\ B$
2. $A \to a\ A$
3. $B \to b\ B$
4. $A \to a$
5. $B \to b$

But : reconnaître le mot "aaabb".

- **Ascendant**

  Lecture : *aaabb*

  $aaabb \overset{4}{\Leftarrow} aaAbb \overset{2}{\Leftarrow} aAbb \overset{2}{\Leftarrow} Abb$



Construction de l'arbre : Exemple

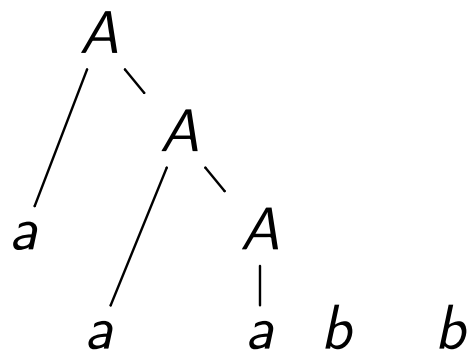
1. $S \to A\ B$
2. $A \to a\ A$
3. $B \to b\ B$
4. $A \to a$
5. $B \to b$

But : reconnaître le mot "aaabb".

- **Ascendant**

  Lecture : *aaabb*

  $aaabb \overset{4}{\Leftarrow} aaAbb \overset{2}{\Leftarrow} aAbb \overset{2}{\Leftarrow} Abb$



Construction de l'arbre : Exemple

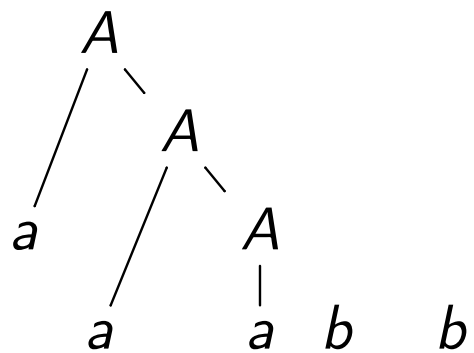
1. $S \to A\ B$
2. $A \to a\ A$
3. $B \to b\ B$
4. $A \to a$
5. $B \to b$

But : reconnaître le mot "aaabb".

- **Ascendant**

  Lecture : *aaabb*

  $aaabb \overset{4}{\Leftarrow} aaAbb \overset{2}{\Leftarrow} aAbb \overset{2}{\Leftarrow} Abb$



Construction de l'arbre : Exemple

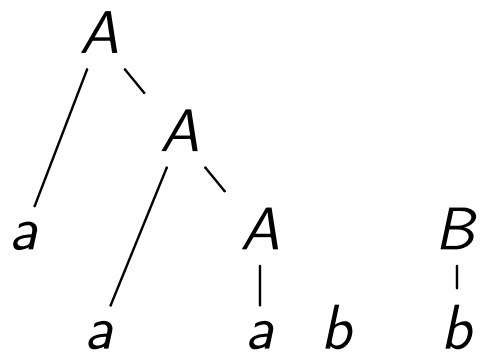
1. $S \to A\ B$
2. $A \to a\ A$
3. $B \to b\ B$
4. $A \to a$
5. $B \to b$

But : reconnaître le mot "aaabb".

- **Ascendant**

  Lecture : *aaabb*

  $aaabb \overset{4}{\Leftarrow} aaAbb \overset{2}{\Leftarrow} aAbb \overset{2}{\Leftarrow} Abb \overset{5}{\Leftarrow} AbB$



Construction de l'arbre : Exemple

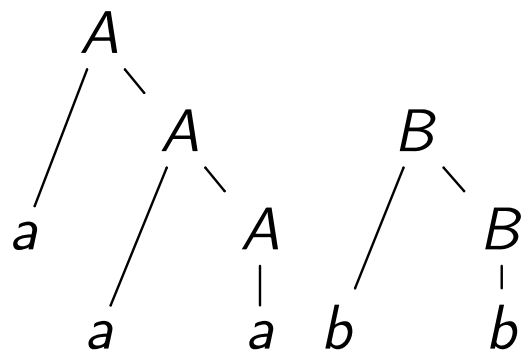
1. $S \to A\ B$
2. $A \to a\ A$
3. $B \to b\ B$
4. $A \to a$
5. $B \to b$

But : reconnaître le mot "aaabb".

- **Ascendant**

  Lecture : *aaabb*

  $aaabb \overset{4}{\Leftarrow} aaAbb \overset{2}{\Leftarrow} aAbb \overset{2}{\Leftarrow} Abb \overset{5}{\Leftarrow} AbB \overset{3}{\Leftarrow} AB$



Construction de l'arbre : Exemple

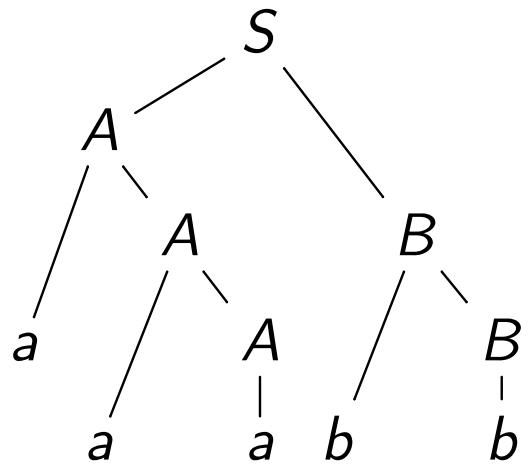
1. $S \to A\ B$
2. $A \to a\ A$
3. $B \to b\ B$
4. $A \to a$
5. $B \to b$

But : reconnaître le mot "aaabb".

- **Ascendant**

  Lecture : *aaabb*

  $aaabb \overset{4}{\Leftarrow} aaAbb \overset{2}{\Leftarrow} aAbb \overset{2}{\Leftarrow} Abb \overset{5}{\Leftarrow} AbB \overset{3}{\Leftarrow} AB \overset{1}{\Leftarrow} S$



Dérivation la plus à droite

Construction de l'arbre : Exemple

1. $S \rightarrow A\ B$
2. $A \rightarrow a\ A$
3. $B \rightarrow b\ B$
4. $A \rightarrow a$
5. $B \rightarrow b$

But : reconnaître le mot "aaabb".

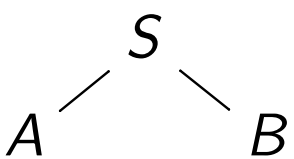
- **Descendant**
  $\underline{S}$
  S

Construction de l'arbre : Exemple

1. $S \rightarrow A\ B$
2. $A \rightarrow a\ A$
3. $B \rightarrow b\ B$
4. $A \rightarrow a$
5. $B \rightarrow b$

But : reconnaître le mot "aaabb".

- **Descendant**
  $\underline{S} \overset{1}{\Rightarrow} \underline{A}B$

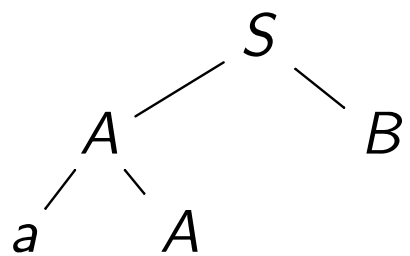


Construction de l'arbre : Exemple

1. $S \to A\ B$
2. $A \to a\ A$
3. $B \to b\ B$
4. $A \to a$
5. $B \to b$

But : reconnaître le mot "aaabb".

- **Descendant**
  $\underline{S} \overset{1}{\Rightarrow} \underline{A}B \overset{2}{\Rightarrow} a\underline{A}B$

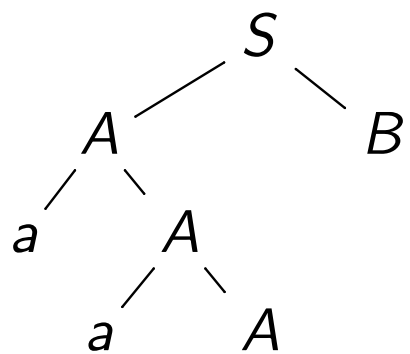


Construction de l'arbre : Exemple

1. $S \rightarrow A\ B$
2. $A \rightarrow a\ A$
3. $B \rightarrow b\ B$
4. $A \rightarrow a$
5. $B \rightarrow b$

But : reconnaître le mot "aaabb".

- **Descendant**
  $\underline{S} \overset{1}{\Rightarrow} \underline{A}B \overset{2}{\Rightarrow} a\underline{A}B \overset{2}{\Rightarrow} aa\underline{A}B$

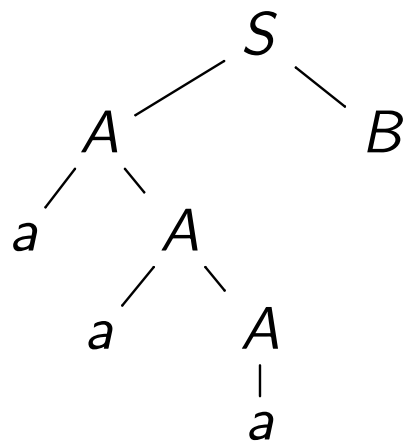


Construction de l'arbre : Exemple

1. $S \to A\ B$
2. $A \to a\ A$
3. $B \to b\ B$
4. $A \to a$
5. $B \to b$

But : reconnaître le mot "aaabb".

- **Descendant**
  $\underline{S} \overset{1}{\Rightarrow} \underline{A}B \overset{2}{\Rightarrow} a\underline{A}B \overset{2}{\Rightarrow} aa\underline{A}B \overset{4}{\Rightarrow} aaa\underline{B}$

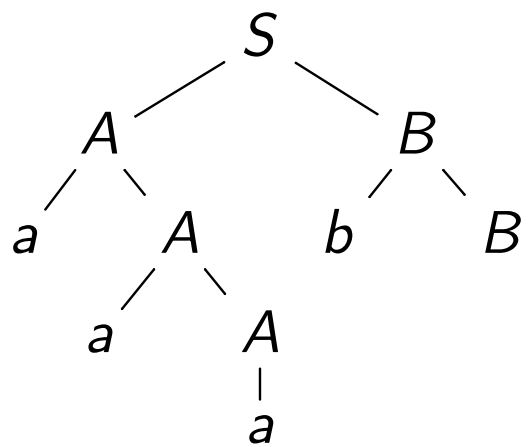


Construction de l'arbre : Exemple

1. $S \to A\ B$
2. $A \to a\ A$
3. $B \to b\ B$
4. $A \to a$
5. $B \to b$

But : reconnaître le mot "aaabb".

- **Descendant**
  $\underline{S} \overset{1}{\Rightarrow} \underline{A}B \overset{2}{\Rightarrow} a\underline{A}B \overset{2}{\Rightarrow} aa\underline{A}B \overset{4}{\Rightarrow} aaa\underline{B} \overset{3}{\Rightarrow} aaab\underline{B}$



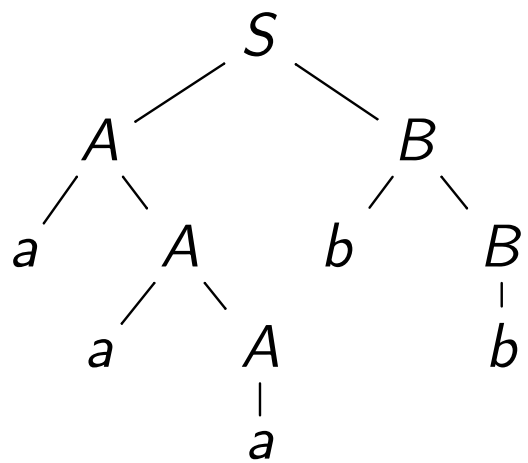
Construction de l'arbre : Exemple

1. $S \to A\ B$
2. $A \to a\ A$
3. $B \to b\ B$
4. $A \to a$
5. $B \to b$

But : reconnaître le mot "aaabb".

- **Descendant**

$$\underline{S} \overset{1}{\Rightarrow} \underline{A}B \overset{2}{\Rightarrow} a\underline{A}B \overset{2}{\Rightarrow} aa\underline{A}B \overset{4}{\Rightarrow} aaa\underline{B} \overset{3}{\Rightarrow} aaab\underline{B} \overset{5}{\Rightarrow} aaabb$$



Dérivation la plus à gauche

Plan

Principe de l'analyse syntaxique descendante

L'analyse syntaxique descendante consiste à construire, en lisant de gauche à droite (Left to right) la dérivation la plus à gauche (Leftmost) correspondant au texte source.
Le principe de l'analyse descendante est donc le suivant :

- Analyser un non-terminal c'est :
  1. "choisir" une règle de production qui le dérive
  2. analyser successivement chacun des symboles de sa partie droite
- Analyser un terminal, c'est le comparer au terminal lu par l'analyseur lexical et le consommer

Exemple

Soit la grammaire :

1. $S' \rightarrow S\$$
2. $S \rightarrow B\ A$
3. $B \rightarrow b\ B$
4. $B \rightarrow \Lambda$
5. $A \rightarrow a\ A$
6. $A \rightarrow \Lambda$

Analyse de "b b a $" : $S' \overset{1}{\rightarrow} \underline{S}\ \$ \overset{2}{\rightarrow} \underline{B}\ A\ \$ \overset{3}{\rightarrow} b\ \underline{B}\ A\ \$ \overset{3}{\rightarrow} b\ b\ \underline{B}\ A\ \$ \overset{4}{\rightarrow} b\ b\ \underline{A}\ \$ \overset{5}{\rightarrow} b\ b\ a\ \underline{A}\ \$ \overset{6}{\rightarrow} b\ b\ a\ \$$

Exemple

Soit la grammaire :

1. $PROG \rightarrow IS\ \$$
2. $IS \rightarrow I\ \ IS$
3. $IS \rightarrow \Lambda$
4. $I \rightarrow\ TYPE\ \ id\ =\ E\ ;$
5. $I \rightarrow\ id\ =\ E\ ;$
6. $I \rightarrow\ print\ \ E;$
7. ...

Analyse de

```
int x = 3;
print x;
```

$PROG \xrightarrow{1} \underline{IS}\ \$ \xrightarrow{2} \underline{I}\ IS\ \$ \xrightarrow{4} \underline{TYPE}\ id\ =\ E\ ;\ IS\ \$ \rightarrow \ldots \rightarrow int\ id\ = 3\ ;\underline{IS}\ \$ \xrightarrow{2} int\ id\ =\ 3\ ;\underline{I}\ IS\ \$ \xrightarrow{6} int\ id\ =\ 3\ ;print\ \underline{E}\ ;\ IS\ \$ \rightarrow \ldots \rightarrow int\ id\ =\ 3\ ;print\ id\ ;\ \underline{IS}\ \$ \xrightarrow{3} int\ id\ =\ 3\ ;print\ id\ ;\ \$$

Premiers et Suivants

Premiers

$$\text{Premiers}(\Lambda) = \{\Lambda\}$$
$$\text{Premiers}(a) = \{a\} \text{ avec } a \in X$$
$$\text{Premiers}(A) = \bigcup_{A \to \gamma \in P} \text{Premiers}(\gamma)$$
$$\text{Premiers}(\alpha\,\beta) = \text{Premiers}(\alpha) \underbrace{\setminus \{\Lambda\} \cup \text{Premiers}(\beta)}_{\text{si } \Lambda \in \text{Premiers}(\alpha)}$$

Suivants

$$\text{Suivants}(A) = \bigcup_{B_i \to \delta_i A \beta_i \in P} \text{Premiers}(\beta_i) \underbrace{\setminus \{\Lambda\} \cup \text{Suivants}(B_i)}_{\text{si } \Lambda \in \text{Premiers}(\beta_i)}$$

Symboles directeurs et grammaire LL(1)

Symboles directeurs

Terminaux permettant de choisir une règle particulière pour dériver un non-terminal en fonction du début des dérivations et des suivants possibles pour ce non-terminal.

$$\text{Directeurs}(A \to \alpha) = \text{Premiers}(\alpha) \underbrace{\setminus \{\Lambda\} \cup \text{Suivants}(A)}_{\text{si } \Lambda \in \text{Premiers}(\alpha)}$$

Grammaire LL(1)

Soit G une grammaire. G est LL(1) ssi elle est algébrique et $\forall$ A, non-terminal tel que $\{A \to \alpha_i\}$ est l'ensemble les règles de production de A :

$$\forall i, j \; i \neq j \Rightarrow \text{Directeurs}(A \to \alpha_i) \cap \text{Directeurs}(A \to \alpha_j) = \emptyset$$

Exercice

Calculer les symboles directeurs de la grammaire suivante :

1. $S' \rightarrow S\$$
2. $S \rightarrow B\ A$
3. $B \rightarrow b\ B$
4. $B \rightarrow \Lambda$
5. $A \rightarrow a\ A$
6. $A \rightarrow \Lambda$

La grammaire est-elle LL(1) ?

Exercice

On a vu que la grammaire des expressions arithmétique ci-dessous est ambiguë.

- $S \rightarrow E$
- $E \rightarrow E + E$
- $E \rightarrow E * E$
- $E \rightarrow id$
- $E \rightarrow num$

1. Cette grammaire est-elle LL(1) ?
2. Sachant que $*$ est prioritaire sur $+$ et que les opérateurs sont associatifs à gauche, donner une grammaire non ambiguë permettant de représenter les expressions arithmétiques.
3. Donner un arbre syntaxique pour $x * y + z$.
4. La nouvelle grammaire est-elle LL(1) ?

Implantation de l'algorithme de l'ADR (Analyse Descendante Récursive)

Intuition : une fonction récursive d'analyse par non terminal qui en fonction du terminal courant aiguille vers la "bonne" règle de production.
Il faut donc :

- autant de procédures d'analyse des non-terminaux que de non-terminaux
- un analyseur lexical : lexical()
- une procédure d'erreur : erreur()
- une procédure d'analyse d'un terminal : accepter(t)

```
accepter(t)={si terminal=t alors terminal=lexical();
                    sinon erreur(); }
```

- une procédure principale : main()

```
main() = { terminal = lexical();
           Analyse de l'axiome;}
```

Exemple d'implantation de l'ADR

1. $S \rightarrow E\ \$ \quad Dir = \{(\}$
2. $E \rightarrow (E\ ES \quad Dir = \{(\}$
3. $ES \rightarrow) \quad Dir = \{)\}$
4. $ES \rightarrow + E) \quad Dir = \{+\}$
5. $ES \rightarrow * E) \quad Dir = \{*\}$
6. ...

```
main() = { terminal = lexical(); analyser_S();}
analyser_S(){
 Si terminal = (  Alors analyser_E(); Accepter($); Sinon erreur();
}
analyser_E(){
 Si terminal = ( Alors accepter( ( ) ; analyser_E(); analyser_ES();
 Sinon erreur();
}
analyser_ES(){
 Si terminal = ) Alors accepter( ) );
 Si terminal = + Alors accepter(+); analyser_E(); accepter( ) );
 Si terminal = * Alors accepter(*); analyser_E(); accepter( ) );
 Sinon erreur();
}
```

Analyseur syntaxique

Il est facile de générer automatiquement des analyseurs syntaxiques pour les grammaires LL(1) et plus généralement LL(k).

Xtext (qui sera utilisé en TP) est lié à ANTLR qui réalise une analyse LL(k) avec backtracking.

Plan

Bilan

Si les contraintes syntaxiques du langage sont données sous forme d'une grammaire LL(*), la phase d'analyse lexicale permet de découper le programme source en tokens qui correspondent au terminaux de la grammaire. La phase d'analyse syntaxique (grâce à un ADR par exemple) permet alors de vérifier que les terminaux sont "dans le bon ordre", soit ordonnés de façon conforme à la grammaire.

TP

- Écriture d'un analyseur lexical avec `jflex`
- Écriture d'un analyseur syntaxique

Quatrième partie IV

Arbre abstrait

Plan

Arbre de dérivation vs Arbre abstrait

- L'arbre de dérivation possède de nombreux nœuds qui ne véhiculent pas d'information.
- La mise au point d'une grammaire (élimination de l'ambiguïté, élimination de la récursivité gauche, ...) nécessite souvent l'introduction de règles dont le seul but est de simplifier l'analyse syntaxique.

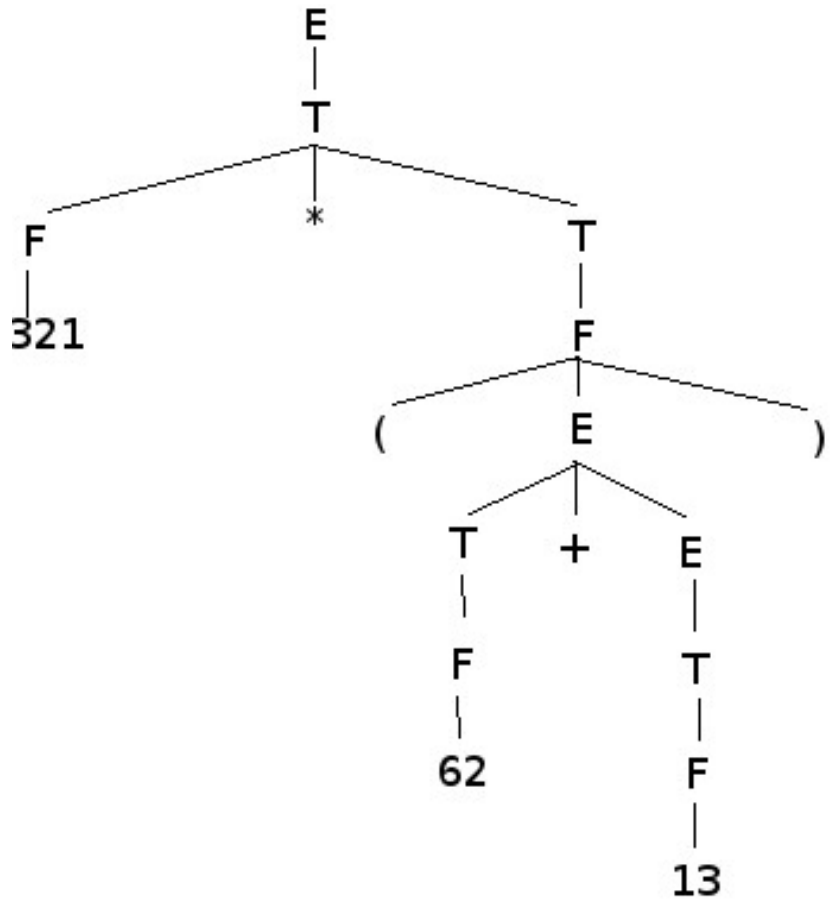
$\Rightarrow$ L'arbre de dérivation contient des informations inutiles et n'est pas toujours simple à exploiter.

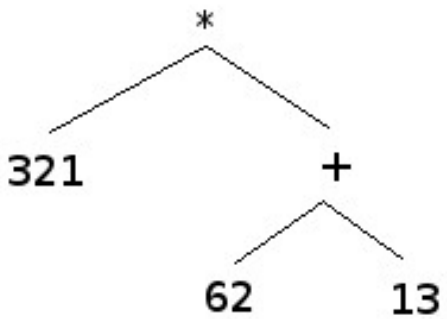
- On va créer un arbre abstrait en ne gardant que les parties nécessaires à l'analyse sémantique et à la génération de code.
- Un arbre abstrait constitue une interface plus naturelle entre l'analyse syntaxique et l'analyse sémantique.

Arbre de dérivation vs Arbre abstrait

- L'arbre de dérivation est unique pour une entrée donnée et une analyse, alors que l'AST est indépendant de l'analyse.
- Les nœuds de l'arbre de dérivation sont imposés par la grammaire, alors que l'on peut choisir le type des nœuds de l'arbre abstrait.

Exemple : Arbre de dérivation vs Arbre abstrait





Plan

Construction de l'arbre abstrait

- L'arbre abstrait est construit lors de l'analyse syntaxique.
- On ajoute aux règles de la grammaire le code destiné à construite l'arbre abstrait.
- Une fois l'arbre abstrait construit, on pourra le parcourir (autant de fois que nécessaire) pour réaliser le traitement sémantique souhaité.

Cinquième partie V

Analyse sémantique : Application à la compilation

Plan

Langage RAT : la grammaire

1. $PROG' \rightarrow PROG\$$
2. $PROG \rightarrow FUN\ PROG$
3. $FUN \rightarrow TYPE\ id\ (\ DP\ )\ \{IS\ return\ E\ ;\}$
4. $PROG \rightarrow id\ \ BLOC$
5. $BLOC \rightarrow \{\ IS\ \}$
6. $IS \rightarrow I\ \ IS_1$
7. $IS \rightarrow \Lambda$
8. $I \rightarrow TYPE\ \ id\ =\ E\ ;$
9. $I \rightarrow id\ =\ E\ ;$
10. $I \rightarrow const\ \ id\ =\ entier\ ;$
11. $I \rightarrow print\ \ E;$
12. $I \rightarrow if\ \ E\ \ BLOC_1\ \ else\ \ BLOC_2$
13. $I \rightarrow while\ \ E\ \ BLOC$
14. $DP \rightarrow \Lambda$
15. $DP \rightarrow TYPE\ id\ DP$
16. $TYPE \rightarrow bool$
17. $TYPE \rightarrow int$
18. $TYPE \rightarrow rat$
19. $E \rightarrow call\ id\ (\ CP\ )$
20. $CP \rightarrow \Lambda$
21. $CP \rightarrow E\ CP$
22. $E \rightarrow [\ E\ /\ E\ ]$
23. $E \rightarrow num\ \ E$
24. $E \rightarrow denom\ \ E$
25. $E \rightarrow id$
26. $E \rightarrow true$
27. $E \rightarrow false$
28. $E \rightarrow entier$
29. $E \rightarrow (\ E\ +\ \ E\ )$
30. $E \rightarrow (\ E\ *\ \ E\ )$
31. $E \rightarrow (\ E\ =\ \ E\ )$
32. $E \rightarrow (\ E\ <\ \ E\ )$

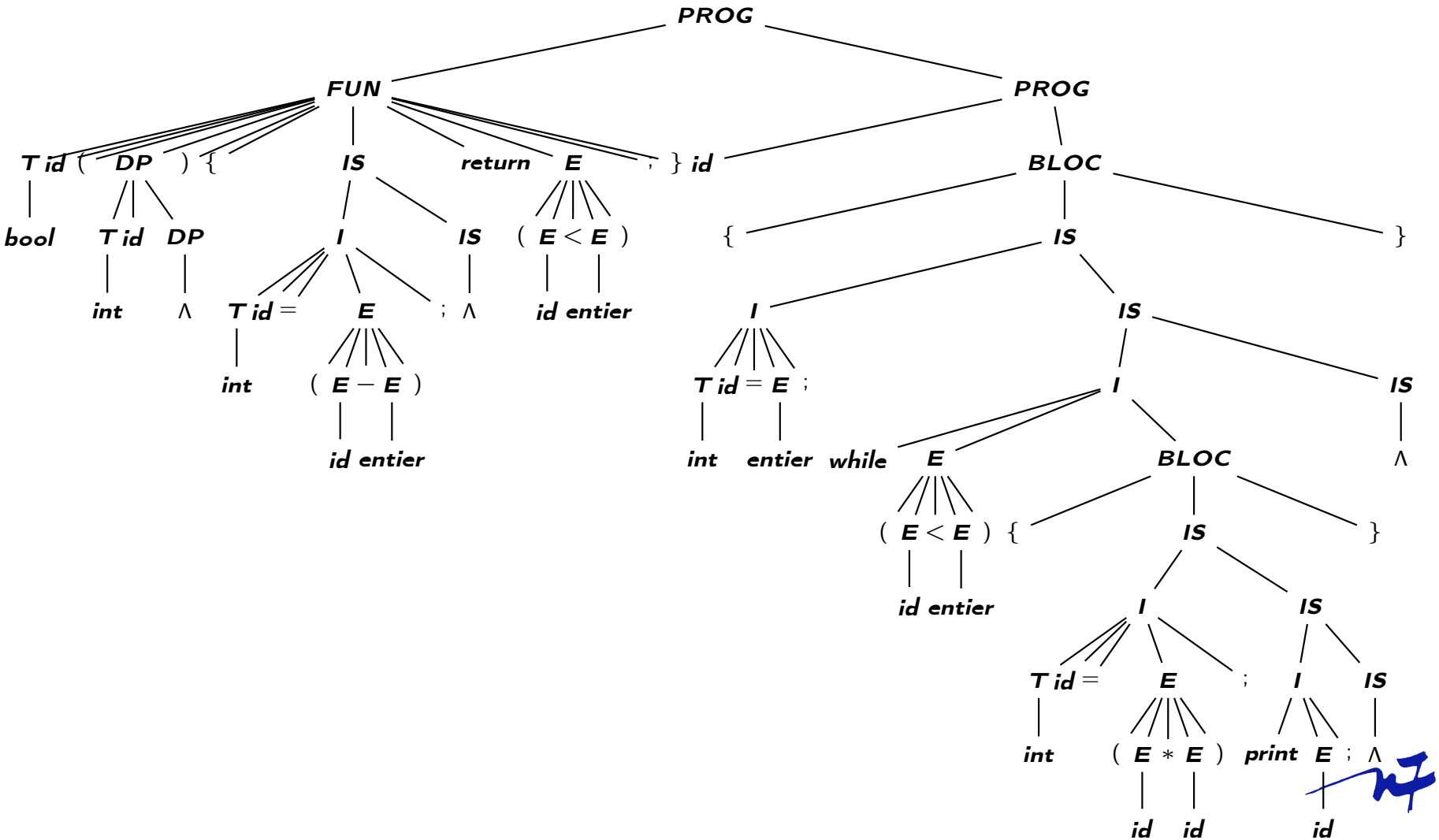
Langage RAT : un exemple

```
bool less ( rat a rat b ){
 return ((num a * denom b )<( num b * denom a ));
}
prog{
 rat a = [3/4];
 rat b = [4/5];
 const n = 5;
 int i = 0;
 while(i<n){
   a=(a+a);
   b=(b*b);
   i=(i+1);
 }
 if(call less(a b)){print a;} else {print b;}
}
```

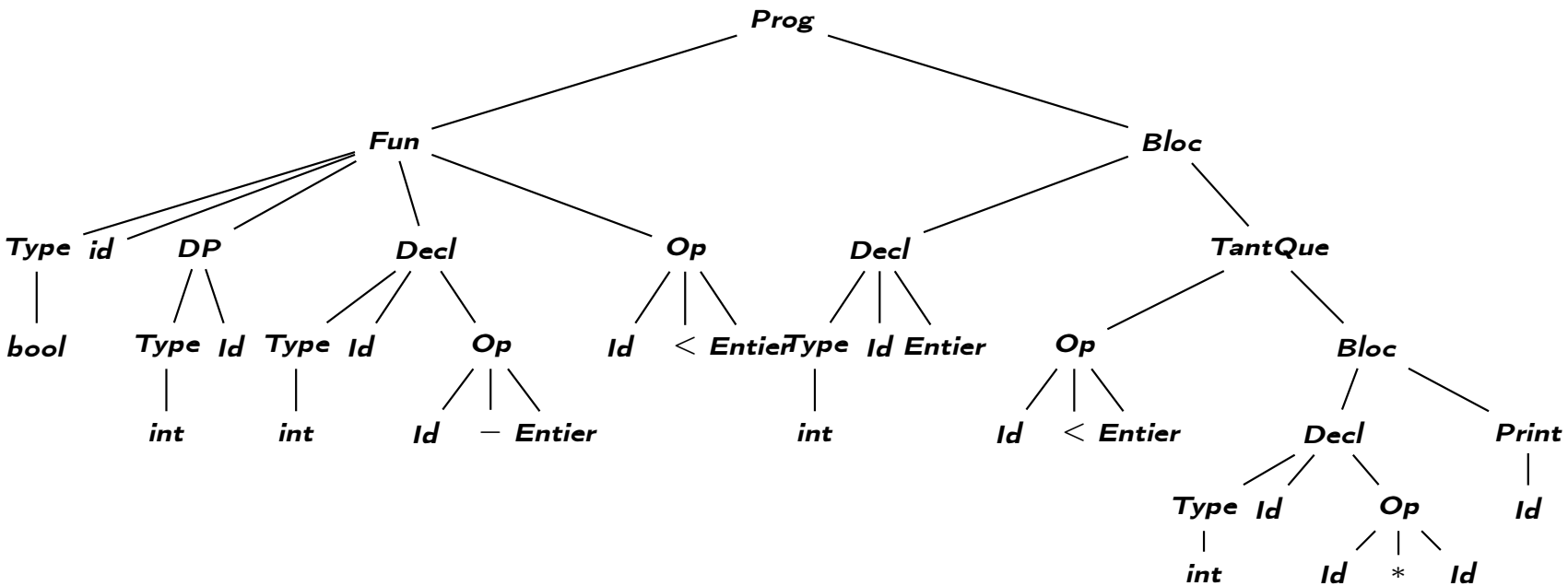
Langage RAT : un exemple simple

```
bool less ( int a ){
 int b = (a-3);
 return (b<0);
}
prog{
 int i = 0;
 while(i<5){
   int j = (i*i);
   print j;
 }
}
```

Arbre de dérivation



Un arbre abstrait possible



Langage RAT : arbre abstrait

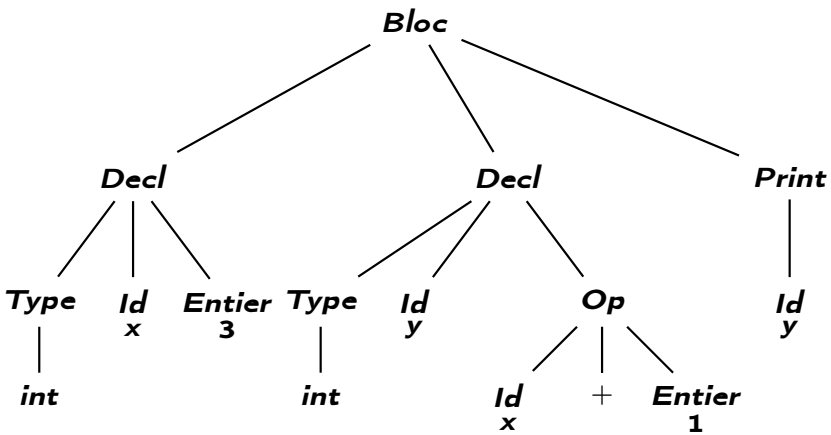
Par rapport à l'arbre de dérivation :

- Suppression des symboles inutiles : (,(,{,},;, ...
- Aplatissement des listes
- Ajout de sous-types pour distinguer les instructions et les expressions $\Rightarrow$ permet l'utilisation de la liaison dynamique

Plan

TDS : retour sur l'exemple

```
int x = 3;
int y = x+1;
print y;
```



TDS : Introduction

- But :
  - Permet de résoudre les identificateurs
  - Fait le lien entre un identificateur et ses caractéristiques
- Les identificateurs :

Type d'identificateur	Informations que l'on veut associer
variable	type, adresse mémoire
constante	type, valeur
procédure / fonction	types des paramètres, type de retour, ...
type	taille, ...
classe	
module	

TDS

Opérations sur la TDS

- Créer la TDS
- Insérer un symbole (nom, information)
- Chercher un symbole (nom $\rightarrow$ information)

Exemple d'implantation

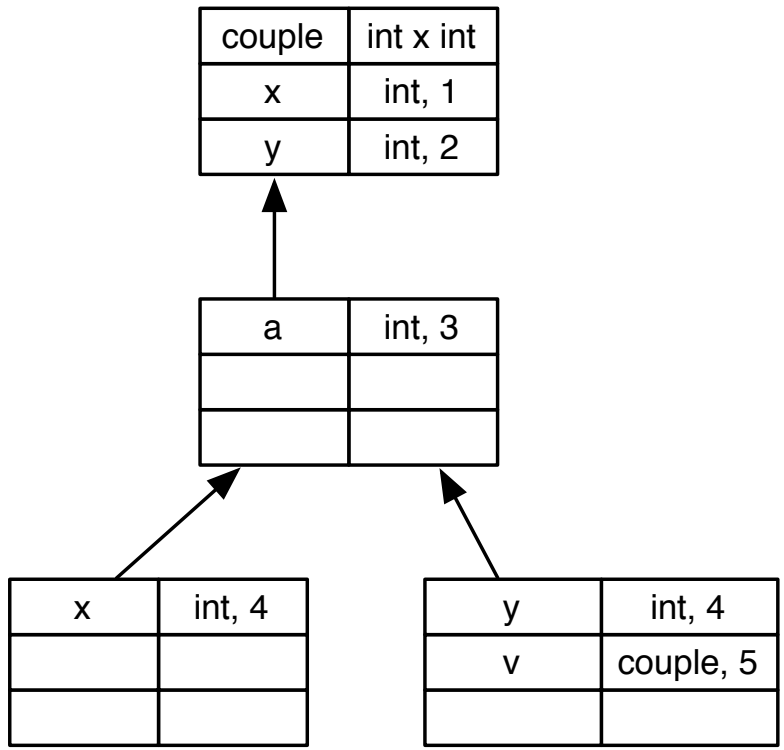
- Tableau
- Liste chainée
- Arbre binaire de recherche
- Table de hachage

TDS $\rightarrow$

Nom	Info

TDS : Gestion des structures imbriquées

```
typedef struct {int data1;
...              int data2;} couple;
int x = 1;
int y = 4;
while (x < y){
   int a = x / y;
   x = x+1;
   if(a < 2){
      int x = 3;
      print x;
   }else{
      int y = 44;
      couple v =[3,4];
      print y;
   }
}
```



⇒ Modification de la recherche : locale ou globale.

Actions à réaliser sur l'AST

- Les actions sémantiques doivent permettre de
  - propager la tds
  - créer une nouvelle table à l'entrée d'un bloc ou d'une déclaration de fonction
  - vérifier la bonne utilisation des identificateurs (pas de double déclaration, pas d'utilisation sans déclaration, pas de modification de constante, ...)
  - stocker la TDS dans l'AST quand nécessaire (pour qu'elle puisse être utiliser lors d'un nouveau parcours de l'AST pour un autre traitement)

$\Rightarrow$ A faire en TD / TP.

Pour aller plus loin...

- Comment faire de la récursivité croisée ?
- Comment passer une fonction en paramètre ?
- Comment faire de l'introspection ?

Typage : un exemple

```
bool foo(int x){
  int y = 1;
  return (x<y);
}
prog{
  int a = 4 ;
  print (call foo(a));
}
```

Il faut vérifier que :

- le type de retour de `foo` est bien `bool`
- le type de x, et le type de y est bien `int` pour savoir que leur comparaison est bien de type `bool` ;
- le type de a est bien `int` pour que l'appel de `foo` soit correct.
- ...

Typage

Type

Un type définit un ensemble de valeurs que peut prendre une donnée, ainsi que les opérateurs qui peuvent être appliquées sur cette donnée.

Contrôle de type

Vérifier que les opérations sont appliquées sur les bons types.

Type de contrôle

- statique : réalisé à la compilation
- dynamique : réalisé à l'exécution (exemple : liaison dynamique)

Système de types

Un système de type

- mélange type de base et types construits.
  - Type de base : atomique, sans structure interne (ou structure non accessible au programmeur), exemple : entiers, booléens, ...
  - Type construits : Construits à partir d'autres types (de base ou construit), et de structures de données comme les tableaux, les ensembles ...
- possède une collection de règles permettant d'associer des expressions de types aux diverses parties d'un programme.

Règle de typage

Sémantique naturelle du typage

- Jugement de typage : $\sigma \vdash e : \tau, \sigma'$
  - σ : environnement de typage (TDS)
  - e : l'expression à typer
  - τ : le type
  - σ' : optionnel, les nouvelles informations à ajouter à l'environnement de typage

Dans la suite : $\tau \neq \mathit{void}$ et $\tau \neq \mathit{erreur}$

Axiomes

- $\sigma :: \{x : \tau\} \vdash x : \tau$
- $\sigma \vdash \quad : \mathit{void}$
- $\dfrac{\sigma \vdash x : \tau}{\sigma :: \{y : \tau'\} \vdash x : \tau}$
- $\sigma \vdash \mathit{true} : \mathit{bool}$
- $\sigma \vdash \mathit{false} : \mathit{bool}$
- $\sigma \vdash \mathit{entier} : \mathit{int}$

Règle de typage du langage RAT

Les cas d'erreur ne seront pas donnés.

Expression (1)

- $\dfrac{\sigma \vdash E_1 : int \quad E_2 : int}{\sigma \vdash [\ E_1\ /\ E_2\ ] : rat}$
- $\dfrac{\sigma \vdash E : rat}{\sigma \vdash num\ E : int}$
- $\dfrac{\sigma \vdash E : rat}{\sigma \vdash denom\ E : int}$
- $\dfrac{\sigma \vdash E : \tau}{\sigma \vdash (E) : \tau}$

Règle de typage du langage RAT

Expression (2)

- $\dfrac{\sigma \vdash E_1 : int \quad E_2 : int}{\sigma \vdash (E_1 + E_2) : int}$
- $\dfrac{\sigma \vdash E_1 : rat \quad E_2 : rat}{\sigma \vdash (E_1 + E_2) : rat}$
- $\dfrac{\sigma \vdash E_1 : int \quad E_2 : int}{\sigma \vdash (E_1 * E_2) : int}$
- $\dfrac{\sigma \vdash E_1 : rat \quad E_2 : rat}{\sigma \vdash (E_1 * E_2) : rat}$
- $\dfrac{\sigma \vdash E_1 : int \quad E_2 : int}{\sigma \vdash (E_1 = E_2) : bool}$
- $\dfrac{\sigma \vdash E_1 : bool \quad E_2 : bool}{\sigma \vdash (E_1 = E_2) : bool}$
- $\dfrac{\sigma \vdash E_1 : int \quad E_2 : int}{\sigma \vdash (E_1 < E_2) : bool}$
- On se limitera à ces signatures.

Règle de typage du langage RAT

Structures de contrôle

- $$\frac{\sigma \vdash E : bool \quad \sigma \vdash BLOC_1 : void \quad \sigma \vdash BLOC_2 : void}{\sigma \vdash if\ E\ BLOC_1\ else\ BLOC_2 : void, \{\}}$$
- $$\frac{\sigma \vdash E : bool \quad \sigma \vdash BLOC : void}{\sigma \vdash while\ E\ BLOC : void, \{\}}$$

Règle de typage du langage RAT

Déclaration / affectation

- $$\frac{\sigma \vdash TYPE : \tau_1 \quad \sigma \vdash E : \tau_2 \quad (estCompatible\ \tau_1\ \tau_2)}{\sigma \vdash TYPE\ id = E : void, \{id : \tau_1\}}$$
- $$\frac{\sigma \vdash id : \tau_1 \quad \sigma \vdash E : \tau_2 \quad (estCompatible\ \tau_1\ \tau_2)}{\sigma \vdash id = E : void, \{\}}$$

Instructions

- $$\frac{}{\sigma \vdash const\ \ id\ \ =\ \ entier : void, \{id : int\}}$$
- $$\frac{\sigma \vdash E : \tau}{\sigma \vdash print\ \ E : void, \{\}}$$

Règle de typage du langage RAT

Déclaration de fonction

- $$\frac{\sigma \vdash \mathit{TYPE} : \tau_1 \quad \sigma \vdash \mathit{DP} : \tau_2, \sigma_p \quad \sigma :: \sigma_p :: \{\mathit{id} : \tau_2 \to \tau_1\} \vdash \mathit{IS} : \mathit{void}, \sigma_l \quad \sigma :: \sigma_p :: \sigma_l \vdash \mathit{E} : \tau_1}{\sigma \vdash \mathit{TYPE}\ \mathit{id}\ (\mathit{DP})\ \{\mathit{IS}\ \mathit{return}\ \mathit{E};\} : \mathit{void}, \{\mathit{id} : \tau_2 \to \tau_1\}}$$
- $$\frac{\sigma \vdash \mathit{TYPE} : \tau_1 \quad \sigma :: \{\mathit{id} : \tau_1\} \vdash \mathit{DP} : \tau_2, \sigma_p}{\sigma \vdash \mathit{TYPE}\ \mathit{id}\ \mathit{DP} : \tau_1 \times \tau_2, \{\mathit{id} : \tau_1\} :: \sigma_p}$$
- $$\frac{\sigma \vdash \mathit{TYPE} : \tau_1}{\sigma \vdash \mathit{TYPE}\ \mathit{id} : \tau_1, \{\mathit{id} : \tau_1\}}$$

Règle de typage du langage RAT

Appel de fonction

- $$\frac{\sigma \vdash id : \tau_1 \to \tau_2 \quad CP : \tau_3 \quad (estCompatible\ \tau_1\ \tau_3)}{\sigma \vdash call\ (id\ CP) : \tau_2}$$
- $$\frac{\sigma \vdash E : \tau_1 \quad CP : \tau_2 \quad \tau_2 \neq void}{\sigma \vdash E\ CP : \tau_1 \times \tau_2}$$
- $$\frac{\sigma \vdash E : \tau_1 \quad CP : void}{\sigma \vdash E\ CP : \tau_1}$$

Règle de typage du langage RAT

Suite d'instructions

- $$\frac{\sigma \vdash IS : void, \sigma'}{\sigma \vdash \{IS\} : void}$$
- $$\frac{\sigma \vdash I : void, \sigma' \quad \sigma :: \sigma' \vdash IS : void, \sigma''}{\sigma \vdash I\ IS : void, \sigma' :: \sigma''}$$

Le programme

- $$\frac{\sigma \vdash FUN : void, \sigma' \quad \sigma :: \sigma' \vdash PROG : void, \sigma''}{\sigma \vdash FUN\ PROG : void, \sigma' :: \sigma''}$$
- $$\frac{\sigma \vdash BLOC : void}{\sigma \vdash id\ BLOC : void}$$

Typage de l'exemple

$$\dfrac{\dfrac{A}{\{\}\vdash bool\ foo(int\ x)\{..\}:void,\{foo:int\to bool\}}\quad\dfrac{B}{\{foo:int\to bool\}\vdash prog\{..\}:void,\{\}}}{\{\}\vdash bool\ foo(int\ x)\{int\ y=1;\ return\ (x<y);\}\ prog\{int\ a=4;\ print(call\ foo(a));\}:void,\{\}}$$

A :

$$\dfrac{\dfrac{}{\{\}\vdash bool:bool}\quad\dfrac{\dfrac{}{\{\}\vdash int:int}}{\{\}\vdash int\ x:int,\{x:int\}}\quad\dfrac{\dfrac{}{\{x:int\}\vdash int:int}\quad\dfrac{}{\{x:int\}\vdash 1:int}}{\{x:int\}\vdash int\ y=1:void,\{y:int\}}\quad\dfrac{\dfrac{\dfrac{}{\{x:int\}\vdash x:int}}{\{x:int,..\}\vdash x:int}\quad\dfrac{}{\{..,y:int\}\vdash y:int}}{\{x:int,y:int\}\vdash (x<y):bool}}{\{\}\vdash bool\ foo(int\ x)\{int\ y=1;\ return\ (x<y);\}:void,\{foo:int\to bool\}}$$

B :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{foo:int\to bool\}\vdash int:int}\quad\dfrac{}{\{foo:int\to bool\}\vdash 4:int}}{\{foo:int\to bool\}\vdash int\ a=4;:void,\{a:int\}}\quad\dfrac{\dfrac{\dfrac{\dfrac{}{\{foo:int\to bool\}\vdash foo:int\to bool}}{\{foo:int\to bool,..\}\vdash foo:int\to bool}\quad\dfrac{}{\{..,a:int\}\vdash a:int}}{\{foo:int\to bool,a:int\}\vdash call\ foo(a);:bool}}{\{foo:int\to bool,a:int\}\vdash (call\ foo(a));:bool}}{\{foo:int\to bool,a:int\}\vdash print(call\ foo(a));:void,\{\}}}{\{foo:int\to bool\}\vdash int\ a=4;\ print(call\ foo(a));:void,\{\}}}{\{foo:int\to bool\}\vdash \{int\ a=4;\ print(call\ foo(a));\}:void,\{\}}}{\{foo:int\to bool\}\vdash prog\{int\ a=4;\ print(call\ foo(a));\}:void,\{\}}$$

Actions à réaliser sur l'AST

- Les actions sémantiques doivent reprendre les règles de typage énoncées précédemment et traiter les cas d'erreur.
- Le type des expressions doivent être stockés dans l'AST quand nécessaire.
  $\Rightarrow$ A faire en TD / TP.

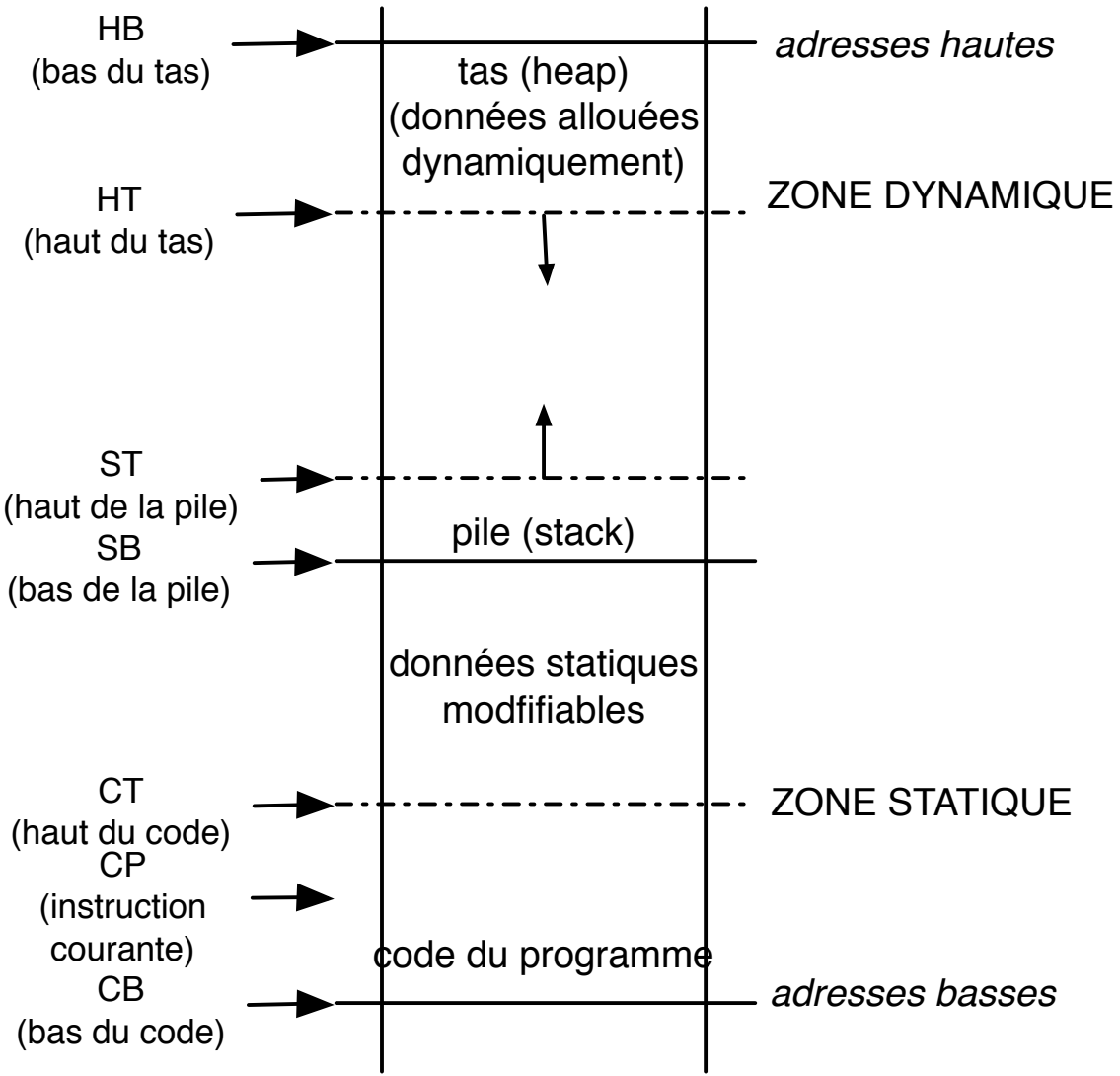
Pour aller plus loin...

- Comment gérer l'héritage de type ?
- Comment ajouter des constructions de type ? (par exemple des couples . . .)
- Comment nommer un type construit ?

Qu'est ce que la mémoire ?

- D'un point de vue utilisateur : grand tableau, dont les indices sont les adresses.
- En pratique : partagé en zone (des adresses hautes vers les adresses basses)
  - le tas (heap) : données allouées dynamiquement par le programme ;
  - la pile (stack) : zone utilisée par les fonctions du programme entre autre pour les variables locales et la sauvegarde du contexte d'appel ;
  - les données allouées statiquement par le programme, c'est le compilateur qui alloue cette zone car contrairement à la précédente, sa taille est connue lors de la compilation (variables globales du programme) ;
  - le code du programme (zone en lecture seulement).
- Remarque : dans le cas de code embarqué, la taille maximale de la mémoire devra être connue.

Représentation graphique



Allocation de la mémoire

- Allocation statique :

```
main(){
   int x = 18;
   int y = 42;
   int z = 60;
   z = x+y;
}
```

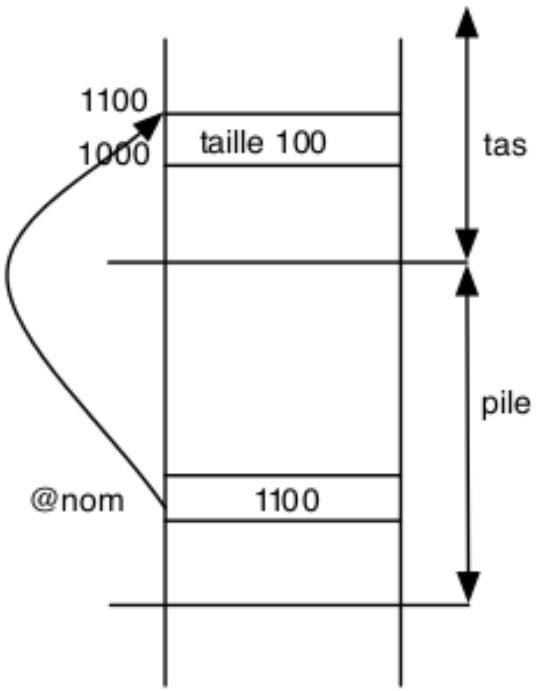
@z	60	d+2
@y	42	d+1
@x	18	d
...		
base		déplacement

Allocation de la mémoire

- Allocation dynamique :
  - Pointeur

```
char *nom;
main(){
    nom = malloc(100);
}
```

A la fin du bloc, la variable disparaîtra, mais pas le "malloc" dont l'adresse pourra avoir été communiquée à d'autres entités.



  - Mémoire nécessaire à l'appel de fonction (cf. plus tard).

Libération de la mémoire

Plusieurs solutions :

- Aucune libération
- Libération automatique : garbage collector, ramasse miettes
  Un procédé qui regarde si certaines zones peuvent être libérées.
  Problème : Ralentit l'exécution
  Deux politiques : ramassage régulier ou à la demande (quand plus de mémoire)
- Le programme lui-même libère la mémoire (free en C)

Machine abstraite à pile

Une forme populaire pour la représentation intermédiaire du code est du code pour une machine abstraite à pile. La machine a une mémoire d'instructions et une mémoire de données séparées et toutes les opérations arithmétiques sont réalisées sur des valeurs de la pile.

Code :

...		← CT
...		
5		← CP
4		
3		
2		
1		
0		← CB

Pile :

...		
5		
4		← ST
3		
2		
1		
0		← SB

Tas :

999		← HB
998		
997		
996		
995		
994		← HT
...		

Déplacement

Informations nécessaire au compilateur :

- variables : adresse (déplacement par rapport à la base de la pile) et taille en mémoire.
- exemple :

```
testType {
  rat a = [1/1];
  rat b = [1/2];
  int c = denom b;
  if (num a > 2) {
    rat a1 = [2/2];
    a = a1;
  }
  else {
       int  i = 0;
  }
  rat d = [3/3];
  int e = 4;
}
```

Donner l'adresse de chacune des variables.

Et les fonctions ?

```
int plus1 (int a int b){
 int c = 1;
 return (a+(b+c));
}
fonction{
 int x = 3;
 int y = 4;
 print call plus1 (x y);
}
```

$\Rightarrow$ Où se trouve a, b et c ?

Enregistrement d'activation

Lors d'un appel de fonction / procédure, nous avons besoin de pouvoir :

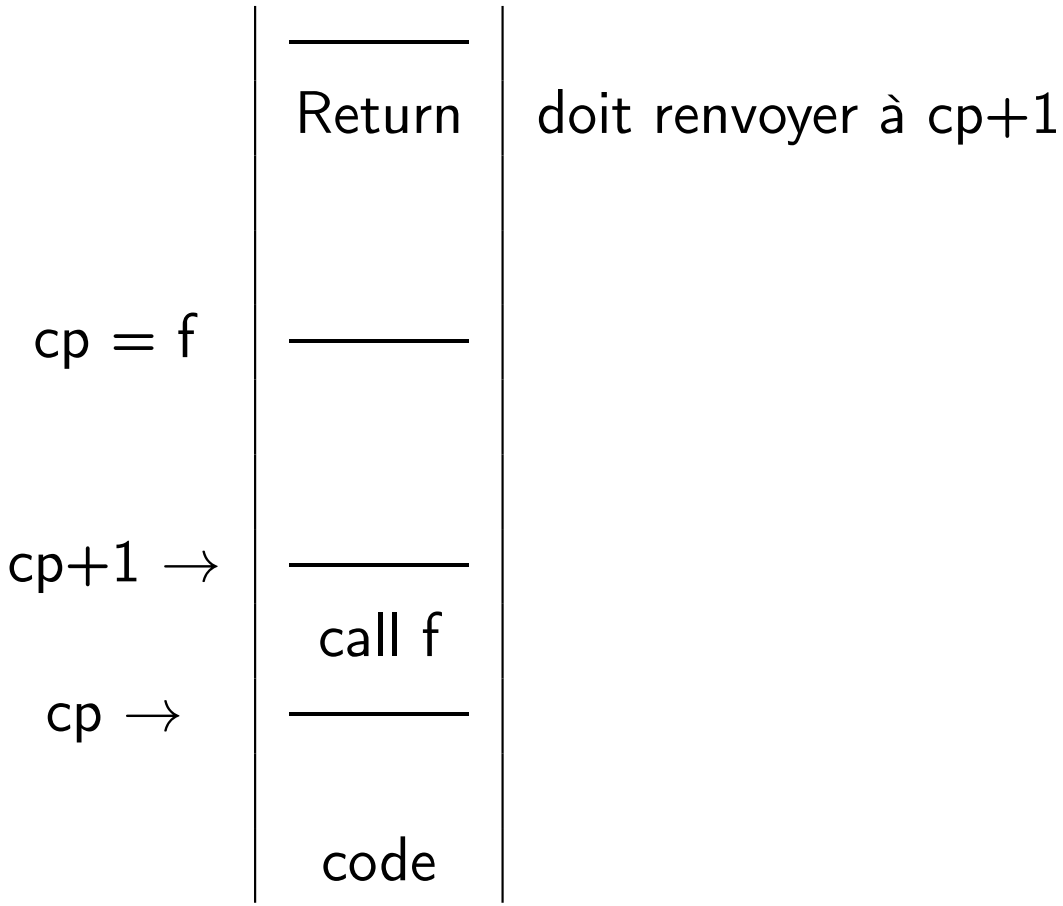
- accéder aux paramètres
- accéder aux variables locales
- restituer l'état courant une fois l'appel fini (si modification)

Solution possible :

- Utiliser la pile pour gérer l'appel de fonction
  Intéressons-nous d'abord à la restitution de l'état au moment de l'appel. Les «choses» qui vont être modifiées sont :
  - le compteur ordinal : la position dans le code. Il faudra connaître l'instruction suivante
  - l'état de la pile : on aura besoin d'un lien dynamique «représentant» l'appelant.

  Ces informations seront stockées dans l'enregistrement d'activation.

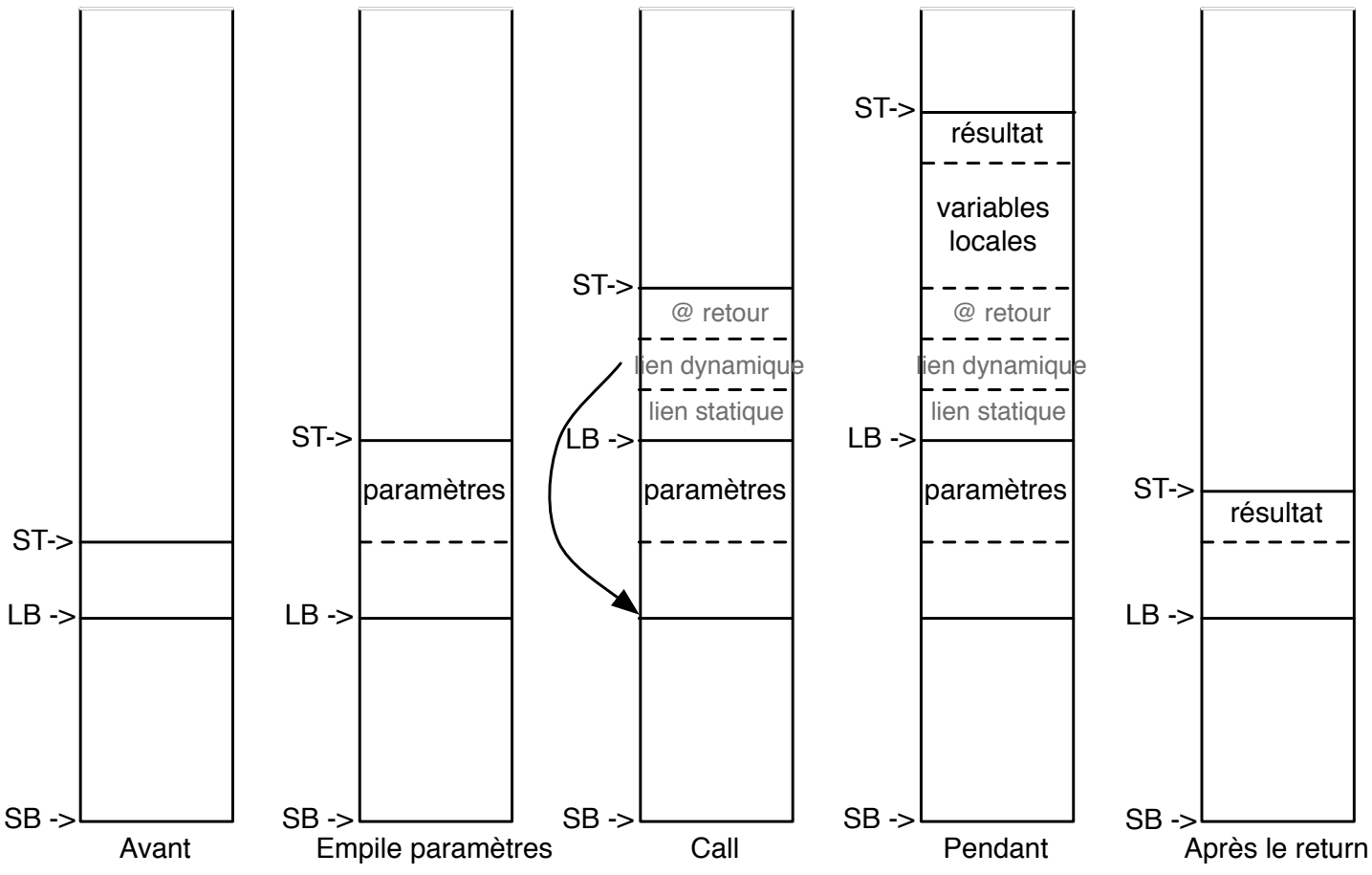
Enregistrement d'activation : l'adresse de retour



Nous avons donc besoin d'un lien vers l'adresse de retour : l'instruction à exécuter une fois la fonction / procédure finie.

Enregistrement d'activation : le lien dynamique

Le lien dynamique va nous permettre de remonter la pile des appels.



Enregistrement d'activation : le lien statique

Le lien statique (troisième information dans l'enregistrement d'activation en TAM) est utile lorsqu'il y a des définitions imbriquées. Le «statique» vient du fait que c'est au moment de sa définition que nous connaissons l'information. En opposition avec le lien dynamique qui représente l'appelant à l'exécution.

Retour sur l'accès aux variables

- Variables du programme principal :
  $\rightarrow$ déplacement positif par rapport à SB (base de la pile).
- Paramètres d'une fonction :
  $\rightarrow$ déplacement négatif par rapport à LB (base de l'enregistrement d'activation).
- Variables locales d'une fonction :
  $\rightarrow$ déplacement positif par rapport à LB.

Actions à réaliser sur l'AST

- Retour sur l'exemple :
  - x : 0[SB]
  - a : -2[LB]
  - b : -1[LB]
    $\Rightarrow$ Attention à l'ordre des paramètres !
  - c : 3[LB] (taille de l'enregistrement d'activation)
  - y : 1[SB]
- Le cas général
  Il faut calculer la base (SB ou LB) et le déplacement par rapport à cette base.
  Il faut ajouter ces informations dans la TDS.
  $\Rightarrow$ A faire en TD / TP

Pour aller plus loin...

- Pas nécessairement besoin de pile pour faire l'appel de fonction, si pas de récursivité.
- La pile du processeur $\neq$ pile des enregistrement d'activation : mais on traite les deux dans une seule.

TAM : Instructions agissant sur la zone de code (et la pile)

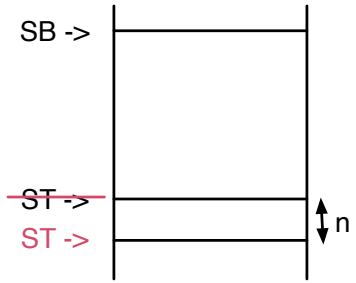
- etiq : déclaration d'une étiquette
- JUMP etiq : déplace la position de CP pour pointer sur l'adresse mise à la place de l'étiquette au moment de l'assemblage
- JUMPIF n etiq : vérifie si la donnée en tête de pile est n et si c'est le cas déplace la position de CP pour pointer sur l'adresse mise à la place de l'étiquette au moment de l'assemblage, consomme la tête de pile.
- JUMPIF n d[r] : vérifie si la donnée en tête de pile est n et si c'est le cas déplace la position de CP pour pointer sur l'adresse d[r], consomme la tête de pile.
- SUBR op : Appel de la procédure simple op, consommation des arguments en sommet de pile (op doit être une procédure prédéfinie dans TAM)

TAM : Instructions agissant sur la zone de code (et la pile)

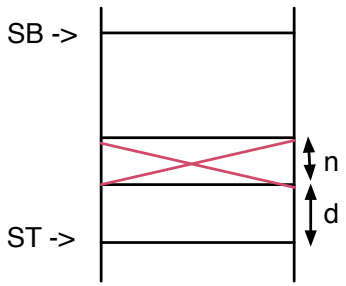
- CALL (r) op : Appel de la procédure complexe op. L'enregistrement d'activation contient 3 champs : le lien dynamique (base d'enregistrement d'activation de l'appelant), lien statique (pour définition imbriquée - passé en paramètre r - on n'en aura à priori pas besoin), l'adresse de l'instruction à exécuter une fois l'appel fini. Un marqueur LB, signale la base de l'enregistrement d'activation (le sommet de pile au moment de l'appel).
- RETURN (r) p : Fin d'un appel de fonction : r est la taille du résultat et p la taille des paramètres. On enlèvera donc un bloc de taille p de la tête de l'ancienne pile et on laissera un bloc de taille r pour les résultats.
- HALT : Arrêt

TAM : Instructions agissant sur la pile

- PUSH n

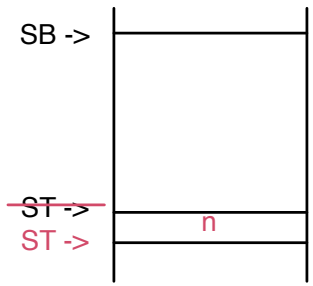


- POP (d) n : On garde les d premiers et on supprime les n suivants

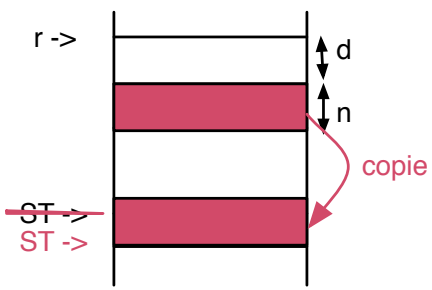


TAM : Instructions agissant sur la pile

- LOADL n : On empile la valeur n

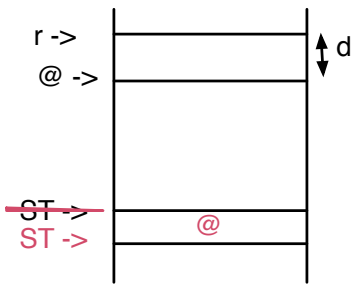


- LOAD (n) d[r] : copie le bloc de taille n qui est à l'adresse d[r] en sommet de pile

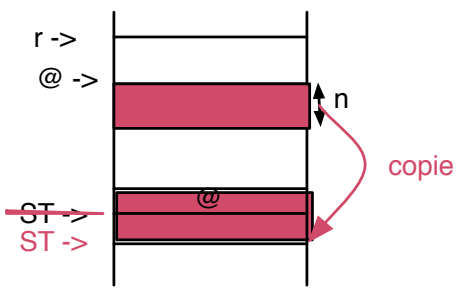


TAM : Instructions agissant sur la pile

- LOADA d[r] : Empile l'adresse d[r] en sommet de pile

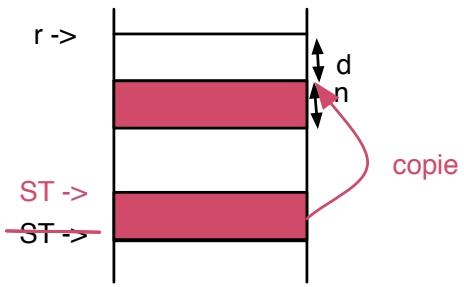


- LOADI (n) : prend l'adresse en sommet de pile, et copie le bloc de taille n de cette adresse en sommet de pile

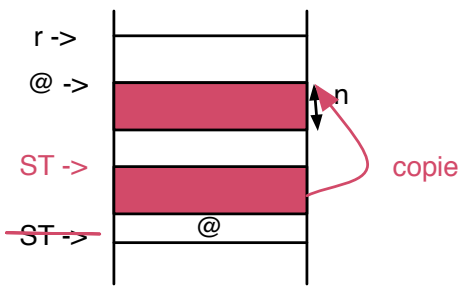


TAM : Instructions agissant sur la pile

- STORE(n) d[r] : déplace le bloc de taille n en sommet de la pile à l'adresse d[r]



- STROREI (n) : prend l'adresse en sommet de pile, et déplace le bloc de taille n en sommet de pile à l'adresse.



Génération de code : RAT → TAM

Exemple simple avec appel de fonction.

```
int plus1(int a int b){
 int c = 1;
 return (a+(b+c));
}
fonction{
 int x = 3;
 int y = 4;
 print call (plus1 x y);
}
```

Génération de code : RAT → TAM

```
main          ;programme principal
PUSH 1              ;place pour x
LOADL 3             ;chargement de l'entier
STORE (1) 0[SB]     ;rangement de 3 à l'adresse de x
PUSH 1              ;place pour y
LOADL 4             ;chargement de l'entier
STORE (1) 1[SB]     ;rangement de 4 à l'adresse de x
LOAD (1) 0[SB]      ;chargement de x
LOAD (1) 1[SB]      ;chargement de y
CALL (-) plus1      ;appel de la fonction
SUBR IOUT           ;appel de l'affichage des entiers
POP (0) 2           ;libération des variables
HALT
```

Génération de code : RAT → TAM

```
plus1         ;fonction plus1
PUSH 1              ;place pour c
LOADL 1             ;chargement de l'entier
STORE (1) 3[LB]     ;rangement de 1 à l'adresse de c
LOAD (1) -2[LB]     ;chargement de a
LOAD (1) -1[LB]     ;chargement de b
LOAD (1) 3[LB]      ;chargement de c
SUBR IAdd           ;appel de l'addition des entiers
SUBR IAdd           ;appel de l'addition des entiers
POP(1) 1            ;libération des variables
RETURN (1) 2        ;fin de la fonction
```

Génération de code : RAT → TAM

Exemple complexe sans appel de fonction

```
prog {
  const a = 8;
  rat x = [6/a];
  int y = (a+1);
  x = (x + [3/2]);
  while (y < 12) {
    rat z = (x * [5/y]);
    print z;
    y = ( y + 1);
  }
}
```

⇒ A faire en TD.

Actions à réaliser sur l'AST

- $\Rightarrow$ A faire en TD / TP.

Plan

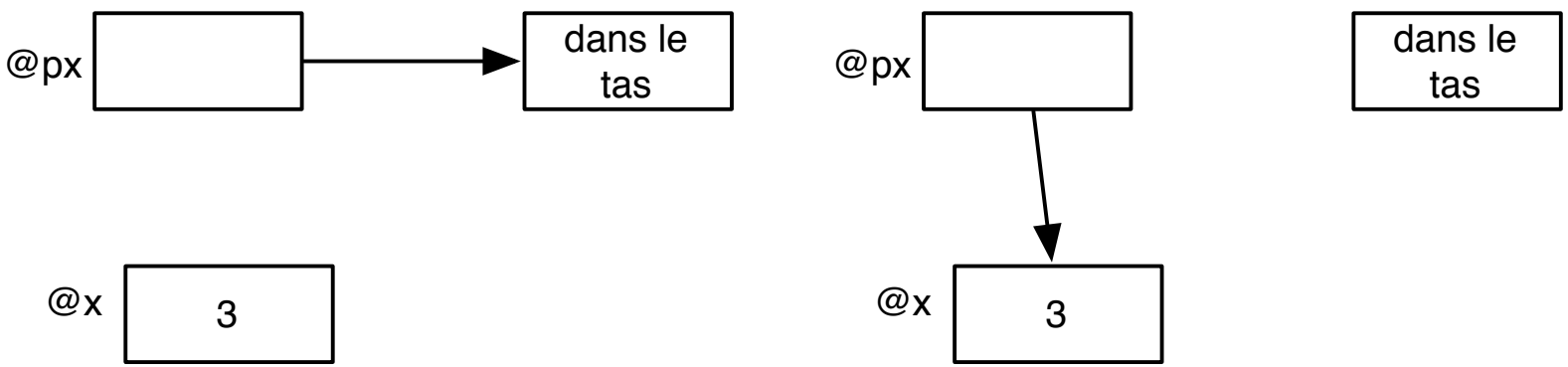
Les pointeurs

Une variable est physiquement identifiée de façon unique par son **adresse**, c'est-à-dire l'adresse de l'emplacement mémoire qui contient sa valeur.
Un **pointeur** est une variable qui contient l'**adresse** d'un autre objet informatique (une « variable de variable» en somme).
La déclaration d'un pointeur se fait selon la syntaxe suivante : type* id ; Cette instruction déclare une variable de nom id et de type pointeur(type) (pointeur sur une valeur de type type).
Exemple : int* x ; déclare une variable x qui pointe sur une valeur de type int.

Les pointeurs

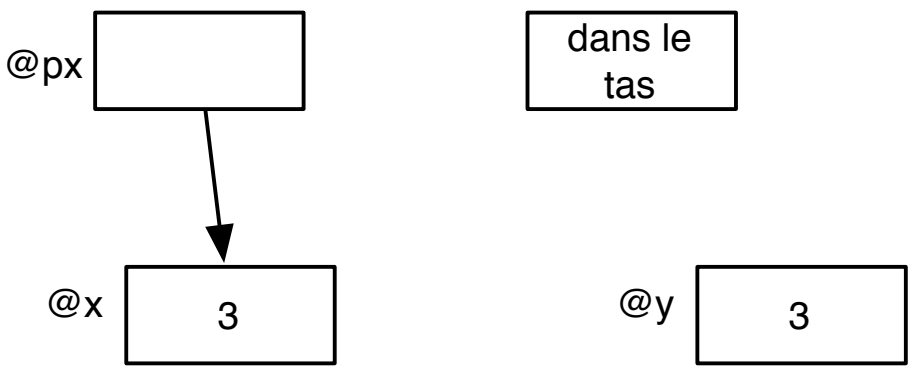
Les deux opérateurs particuliers en relation avec les pointeurs sont : & et *.

- & est l'opérateur qui **retourne l'adresse mémoire d'une variable**
  Si x est de type t alors &x est de type Pointeur(t)
  Exemple :
  int * px = new int ;
  int x = 3 ;
  px = &x ;



Les pointeurs

- * est l'opérateur qui **retourne la valeur pointée par une variable pointeur**.
  Si x est de type Pointeur (t) alors *x est de type t
  Exemple :
  int * px = new int ;
  int x = 3 ;
  px = &x ;
  int y = *px ;



Les pointeurs

1. Quelle(s) règles(s) faut-il modifier ou ajouter pour compléter l'introduction des pointeurs dans le langage ?
2. Ecrire quelques exemples de programmes manipulant des pointeurs. Dessiner l'évolution de la mémoire.
3. Faut-il modifier la gestion de la table des symboles ? Si oui, comment ?
4. Modifier le contrôle de type.
5. Proposer la traduction en TAM de quelques exemples d'utilisation des pointeurs.

   On supposera l'existence des instructions TAM suivantes, permettant de manipuler des adresses :
   - Empiler une adresse : LOADA d[r]
   - Empiler n mots à partir de l'adresse laissée en sommet de pile : LOADI (n)
   - Ecrire n mots de la pile à l'adresse laissée en sommet de pile : STOREI (n)
   - Allocation de mémoire : SUBR Malloc (réserve dans la tas une zone de la taille laissée en sommet de pile, l'adresse est laissée en sommet de pile à la place du tas).
6. Donner les actions sémantiques pour la génération de code.

Plusieurs fichiers / librairies

- Compilation de plusieurs fichiers (sans les lier) $\Rightarrow$ plusieurs fichiers objets regroupés en librairies ou archives.
- L'éditeur de liens statique prend tous les fichiers émis et fabrique l'exécutable en les mettant les uns derrière les autres et en résolvant les références symboliques entre ces fichiers.
  $\Rightarrow$ Soucis : lors de l'utilisation d'une librairie, on copie le code des fonctions de la librairie dans l'exécutable.
- Chargement dynamique : le chargement en mémoire du code des bibliothèques est retardé à l'exécution. On remplace l'édition de liens statique par l'ajout de code qui permet d'aller chercher les information sur les symboles non résolus.
  $\Rightarrow$ Soucis : compilation et exécution doivent être fait dans des environnements compatibles.

Pour aller plus loin...

- Comment traiter les mécanismes d'exportation ?
- Comment traiter la surcharge ?
- Comment traiter la déclaration en avant ?
- Comment traiter les exceptions ?
- Comment traiter les modules ?
- ...

Sixième partie VI

Conclusion

Conclusion

Pour écrire un compilateur pour un langage simple à l'aide d'un schéma de traduction, il faut :

1. Donner la grammaire du langage ;
2. Ecrire l'analyseur lexical des terminaux de la grammaire ;
3. Coupler l'analyseur lexical avec un analyseur syntaxique qui vérifie que le programme est bien conforme à la grammaire ;
4. Compléter l'analyseur syntaxique pour en faire un analyseur sémantique qui permet le traitement des identificateurs, la vérification de type et la génération de code (en se basant sur le placement mémoire des variables).

Conclusion

- Si la grammaire est LL(*) :
  - L'analyseur syntaxique peut être réalisé par une ADR ;
  - L'analyseur syntaxique peut être engendré ;

  $\Rightarrow$ Utilisation de générateur de compilateur.
- Si le langage est complexe, toutes les phases de la compilation seront nécessaires (linéarisation, sélection d'instructions, allocation de registres, optimisation, édition de liens, . . .).