Traduction des langages

Aurélie Hurault, Philippe Quéinnec
hurault@enseeiht.fr, queinnec@enseeiht.fr

Plan

I. Introduction

II. Analyse lexicale : Rappels et compléments

Plan

III. Analyse syntaxique

IV. Grammaires attribuées

V. Arbre syntaxique abstrait (AST)

Plan

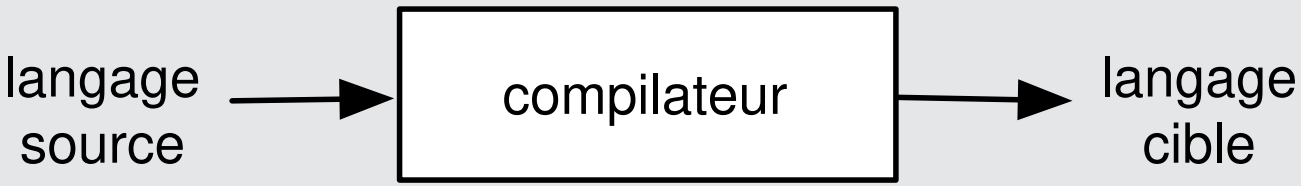
Première partie I

Introduction

Objectif du cours

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

Qu'est ce qu'un compilateur ?



Pourquoi ?

- Comprendre le fonctionnement des outils utilisés (compilateur, IDE, analyseur de code. . .)
- Comprendre les limites de ces outils ou des langages qu'ils manipulent
- Savoir écrire un analyseur pour un langage simple (p.e. fichier de configuration)
- Savoir écrire un traducteur d'un langage dans un autre (p.e. portage de configuration)
- Faire le lien entre le code haut niveau et son exécution bas niveau sur un processeur
- En bonus, pratiquer la programmation fonctionnelle sur un gros problème

Introduction à la compilation

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

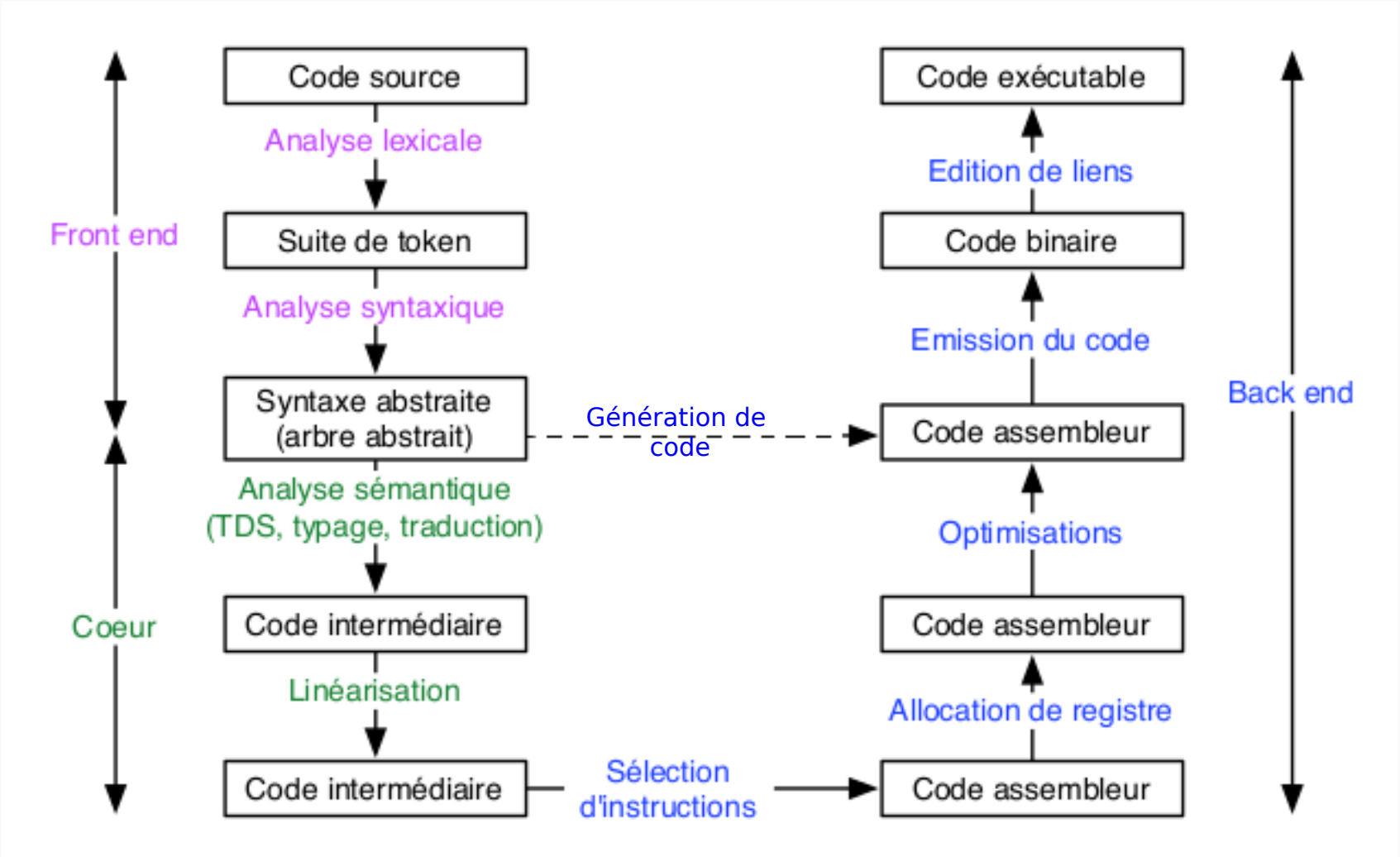
Trois langages

Distinguer 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



Du source à l'exécutable

Exemple fil rouge

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
```

Du source à l'exécutable

Front-end

- Lié au langage source
- Analyse lexicale : découpe le code source en suite de terminaux
- Analyse syntaxique : structure le code source
  - Le code source doit respecter une syntaxe particulière
  - Cette syntaxe est décrite à l'aide d'une grammaire

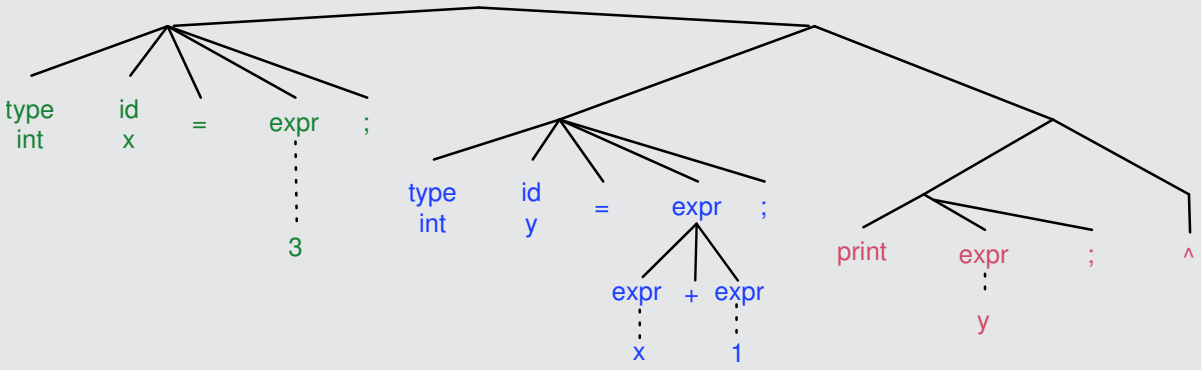
Front-end - Exemple

- 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

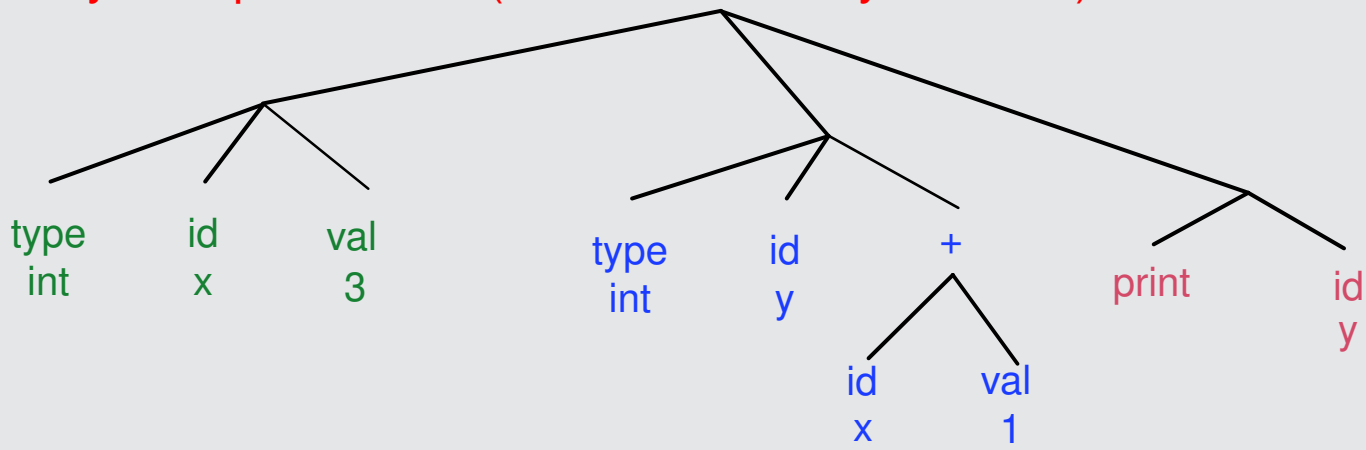
Du source à l'exécutable

Cœur : 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.

Cœur : Analyse sémantique - Exemple

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



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

$\Rightarrow$ Analyse sémantique

Cœur : Analyse sémantique - Exemple suite

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

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

Du source à l'exécutable

Exemple de code intermédiaire

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

Cœur : 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.

Cela peut nécessiter d'introduire des variables intermédiaires ou temporaires.

Cœur : Linéarisation - Exemple

- 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);
```

Du source à l'exécutable

Back-end

- 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 machine qui la réalisent.
  - Si plusieurs séquences d'instructions possibles, il faut faire un choix.

Back-end - Exemple

- 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'instructions (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 - 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.

Back-end - Allocation de registre - Exemple

- 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
```

Du source à l'exécutable

Back-end - Optimisations

- Suppression de code mort.
- Optimisation des boucles.
- Factorisation.
- ...

Back-end - Émission de code

- Le code peut alors être émis dans un fichier.

Du source à l'exécutable

Exemple - Code assembleur après optimisation

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

Back-end - Edition de liens

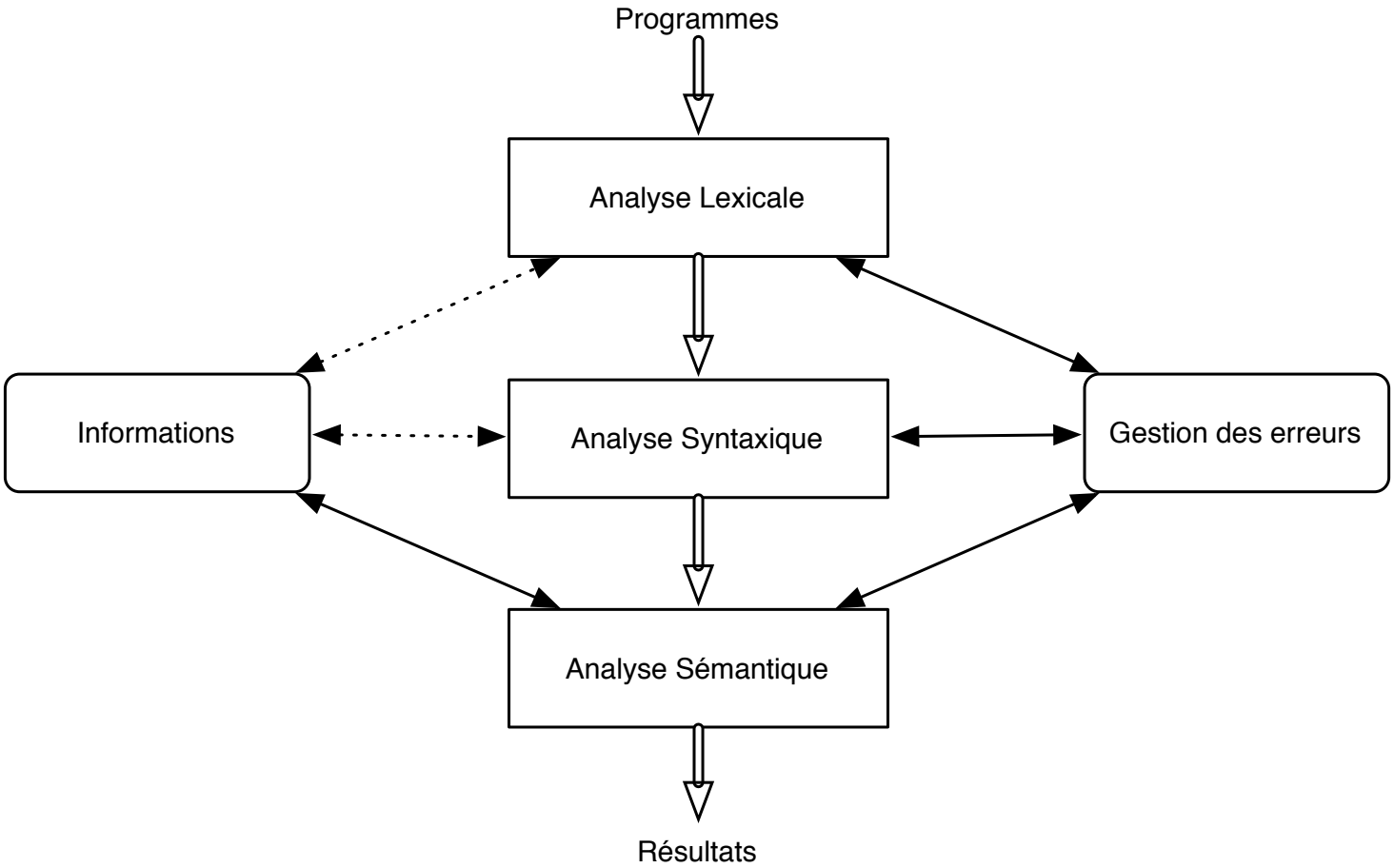
- Quand un programme est réparti sur plusieurs fichiers et qu'il y a des références entre eux, il faut pouvoir accéder aux informations des différents fichiers.
- C'est un autre programme, dit éditeur de liens, qui prend tous les fichiers émis et fabrique l'exécutable. L'éditeur de lien s'occupe essentiellement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

Les points abordés

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, ...)
  - Arbre de dérivation
- Arbre syntaxique abstrait
- 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

♪♪♪

Rappels et compléments : Automates et expressions régulières

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 $L_1, L_2 \in \text{RAT}$, 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

Syntaxe

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

Sémantique

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

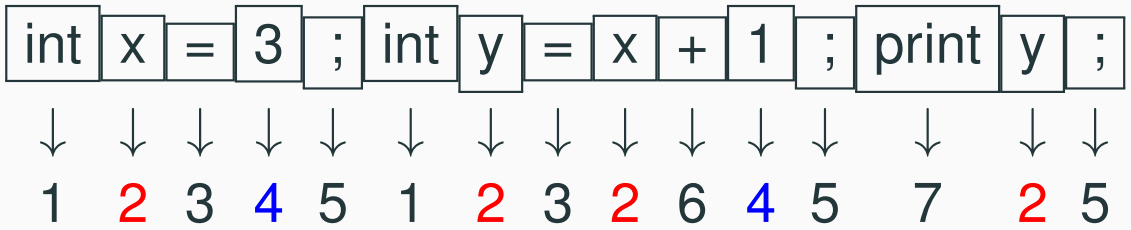
Il y a équivalence entre :

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

Compléments : Analyse lexicale

Analyse lexicale

- Découpe un flux continu en terminaux (ou unité lexicale : code qui correspond à un symbole).
- Un terminal est décrit 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

Outils

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

Ocamllex

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

```
{
  ... code OCaml arbitraire ...
}
rule f1 = parse
 | regexp1 { action1 }
 | regexp2 { action2 }
 | ...
and f2 = parse
  ...
and fn = parse
  ...
{
  ... code OCaml arbitraire ...
}
```

- Les sections "code OCaml arbitraire" sont recopiées en tête et fin du fichier engendré.
- Dans la partie entête : ouverture de modules, déclaration d'exceptions ou fonctions utilisées dans les actions.
- Règles lexicales : Associer une action à une expr. régulière.

Un exemple avec Ocamllex

```
{
 open Parser
 exception Error of string
}

(* Définition de macros pour les expressions régulières *)
let letter = ['A'-'Z']

(* Règles lexicales *)
rule token = parse
| [' ' '\t' '\n'] (* traitement des blancs *)
    { token lexbuf }
| ['0'-'9']+ as i
    { INT (int_of_string i) }
| "XXXX"
    { XXXX }
| (letter|"-")+ as n
   { NAME n }
| eof
    { EOF }
| _
{ raise (Error ("Unexpected char: "^(Lexing.lexeme lexbuf)^
                " at "^(string_of_int (Lexing.lexeme_start lexbuf))^
                "-"^(string_of_int (Lexing.lexeme_end lexbuf)))) }
```

Bilan

Bilan

1. La phase d'analyse lexicale permet de découper le programme source en tokens qui correspondent aux terminaux de la grammaire.

Troisième partie III

Analyse syntaxique

♪♪♪

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

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

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

  $\}$

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

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

Dérivation

Exemple de dérivation

- Soit la grammaire $G = (\{S\}, \{a, b, c\}, P, S)$ avec les productions P :
  1. $S \to a\ S\ b$
  2. $S \to c$
- Dérivation : $S \Rightarrow_1 aSb \Rightarrow_1 aaSbb \Rightarrow_2 aacbb$
- Langage : $L(G) = \{a^n c b^n \mid n \geq 0\}$

Note : les grammaires arbitraires sont aussi puissantes que n'importe quel langage de programmation. C'est trop pour un construire facilement un analyseur $\to$ on va considérer une classe restreinte de grammaires, mais encore suffisamment expressives.

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.

Grammaire algébrique

Exemples de grammaire algébrique

- La grammaire précédente (pour l'exemple de la dérivation).
- La grammaire des expressions : $(\{E\}, \{+, *, id\}, P, E)$ avec P :
  1. $E \rightarrow E + E$
  2. $E \rightarrow E * E$
  3. $E \rightarrow id$
- Un extrait de la grammaire (simplifiée) des instructions C :
  1. $Bloc \rightarrow \{\ ListeInst\ \}$
  2. $ListeInst \rightarrow Inst\ ListeInst$
  3. $ListeInst \rightarrow \Lambda$
  4. $Inst \rightarrow id = E\ ;$
  5. $Inst \rightarrow if\ (\ E\ )\ Bloc$
  6. $Inst \rightarrow if\ (\ E\ )\ Bloc\ else\ Bloc$

Grammaire EBNF

EBNF (extended Backus-Naur form)

Opérateurs simplifiant l'écriture :

opérateur	forme	équivalence
$\mid$: choix	$X \to \alpha \mid \beta$	$X \to \alpha$ $X \to \beta$
? : option	$X?$	X' avec $X' \to \Lambda \mid X$
$*$: répétition optionnelle	$X*$	X' avec $X' \to \Lambda \mid X\ X'$
$+$: répétition non vide	$X+$	X' avec $X' \to X \mid X\ X'$

Exemple

1. *Bloc* $\to$ { *Inst*$*$ }
2. *Inst* $\to$ *id* = *E* ;
$\mid$ *if* (*E*) *Bloc ElsePart*?
3. *ElsePart* $\to$ *else Bloc*

Arbre de dérivation

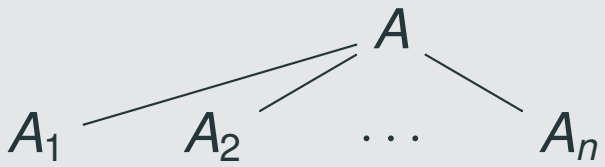
Arbre de dérivation

Création de l'arbre de dérivation

Objectif : représenter la structure du mot induite par les règles de production lors d'une dérivation.

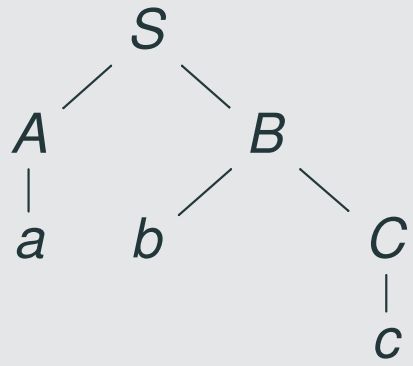
Racine = l'axiome, nœuds = non-terminaux, feuilles = terminaux, branches = règles de production :

utilisation de $A \to A_1 A_2 .. A_n$:



Exemple

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



$abc \in L(G)$:

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

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

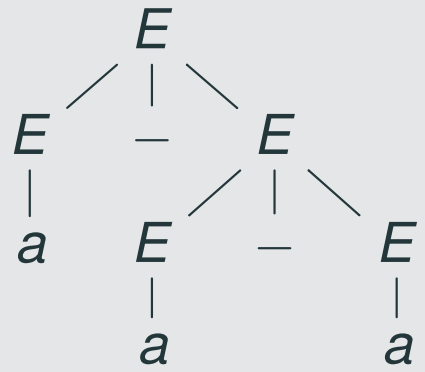
$\underline{S} \overset{1}{\Rightarrow} A\underline{B} \overset{3}{\Rightarrow} Ab\underline{C} \overset{4}{\Rightarrow} \underline{A}bc \overset{2}{\Rightarrow} abc$

Ambiguïté

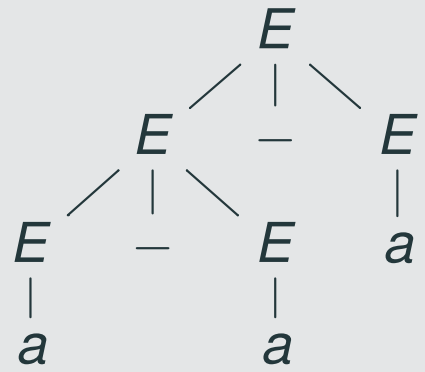
Deux arbres pour un même mot

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

$a - a - a \in L(G)$:



ou



C'est gênant car l'arbre peut être interprété structurellement :
$a - a - a$ est équivalent à $a - (a - a)$ ou $(a - a) - a$?

Élimination de l'ambiguïté

Élimination de l'ambiguïté

Ici, on peut éliminer l'ambiguïté :

$$G' : \begin{cases} 1. & E & \to T\ a \\ 2. & T & \to a - T \\ 3. & T & \to \Lambda \end{cases} \qquad G'' : \begin{cases} 1. & E & \to E - T \\ 2. & E & \to a \\ 3. & T & \to a \end{cases}$$

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

Langage inhéremment ambigu

- Il existe des langages algébriques inhéremment ambigus, i.e. dont toutes les grammaires sont ambiguës.
- Exemple : $L = \{a^n b^p c^q \mid n = p \text{ ou } p = q\}$ 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 $G = (\{A\}, \{a, b\}, P, A)$ avec P :

1. $A \to a\ A\ b$
2. $A \to A\ A$
3. $A \to b\ A\ a$
4. $A \to \Lambda$

Questions **Correction ♪♪♪**

1. Donner des mots de $L(G)$ de taille 0, 2, 4.
2. Tous les mots commençant par a se terminent-ils par b ?
3. Intuitivement, quel est le langage engendré par G ?
4. Donner pour *aabbbaab*, une dérivation la plus à droite [1], une dérivation la plus à gauche [2] et un arbre de dérivation.
5. G est-elle ambiguë ?

1. Quand vous avez le choix entre plusieurs terminaux à dériver, vous dérivez celui qui est le plus à droite.
2. Quand vous avez le choix entre plusieurs terminaux à dériver, vous dérivez celui qui est le plus à gauche.

Exercice

Questions **Correction ♪♪♪**

Pour chacun des langages ci-dessous, donner une grammaire algébrique qui l'engendre :

1. le langage des palindromes sur $\{a, b\}$
2. $\{a^n b^p \mid n \geq p \geq 0\}$
3. le langage des expressions bien parenthésées, par exemple (([•][(•)(•)])(•))

Analyseur syntaxique

Analyseur syntaxique

Définition

- 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 efficaces 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

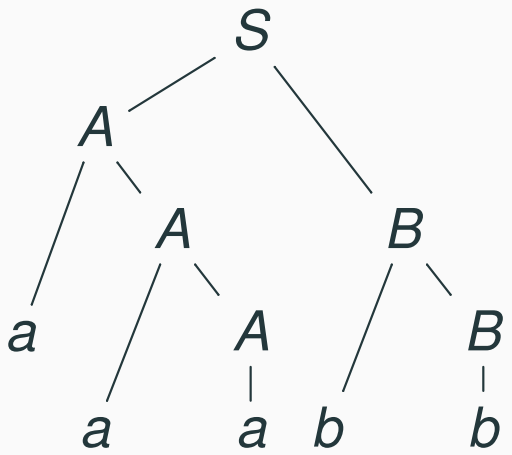
1. $S \rightarrow A\ B$
2. $A \rightarrow a\ A$
3. $B \rightarrow b\ B$
4. $A \rightarrow a$
5. $B \rightarrow b$

But : reconnaître le mot "aaabb".

- **Ascendant**

  Lecture : *aaabb*

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



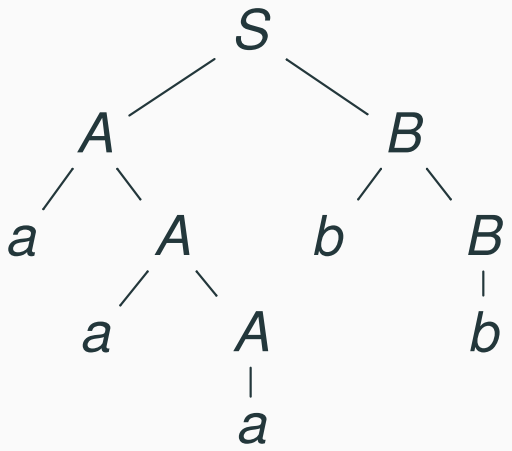
Dérivation la plus à droite

Construction de l'arbre : Exemple

1. $S \to A\ B$
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

Analyse syntaxique

À partir d'une grammaire algébrique, il existe des outils pour générer automatiquement des analyseurs syntaxiques ascendants ou descendants.

Outils

- Analyseur ascendant
  - Monde C : Yacc, Bison
  - Monde Java : Cup
  - Monde OCaml : Ocamlyacc, **Menhir**
  - …
- analyseur descendant
  - Monde Java : ANTLR
  - …

Menhir

- Outil d'analyse syntaxique dans le monde OCaml.
- Format des fichiers

```
Déclarations
%%
Règles de production et traitement
%%
Code OCaml
```

- La partie déclaration peut / doit contenir :
  - du code qui sera recopié dans le fichier généré (ouverture de modules, déclarations d'exceptions, déclarations de fonctions. . .)
  - les tokens (terminaux) de la grammaire
  - des déclarations d'associativité
  - le type des actions associées à un terminal (l'analyse syntaxique est usuellement couplée à un traitement)
  - l'axiome de la grammaire

Grammaire du fichier de statistiques sur les prénoms

```
Main -> Ligne $                    Ligne -> int name int int
Main -> Ligne Main                 Ligne -> int name XXXX int
```

Un exemple avec Menhir (TP2 de Programmation Fonctionnelle)

```
(* sans EBNF *)
%token INT
%token NAME
%token XXXX
%token EOF

(* Axiome *)
%start main

%%
main:
| ligne EOF  {()}
| ligne main {()}

ligne :
| INT NAME INT INT  {()}
| INT NAME XXXX INT {()}
```

```
(* avec EBNF *)
%token INT
%token NAME
%token XXXX
%token EOF

(* Axiome *)
%start main

%%
main: ligne+ EOF  {()}


ligne :
| INT NAME INT INT  {()}
| INT NAME XXXX INT {()}
```

Bilan

Bilan

1. La phase d'analyse lexicale permet de découper le programme source en tokens qui correspondent aux terminaux de la grammaire.
2. La phase d'analyse syntaxique permet alors de vérifier que les terminaux sont "dans le bon ordre", soit ordonnés de façon conforme à la grammaire.

Quatrième partie IV

Grammaires attribuées

Principe

Soit une grammaire avec des règles de productions de la forme :
$A \rightarrow X_1 ... X_n$
On va :

- Associer des informations aux symboles A, X_1, ..., X_n
  Ce sont les attributs sémantiques, ils contiennent les informations concernant la traduction attribuée aux différents symboles de la grammaire.
- Associer des traitements aux règles de productions
  Ce sont les actions sémantiques. Elles permettent de calculer la valeur des attributs.

Exemple introductif (1)

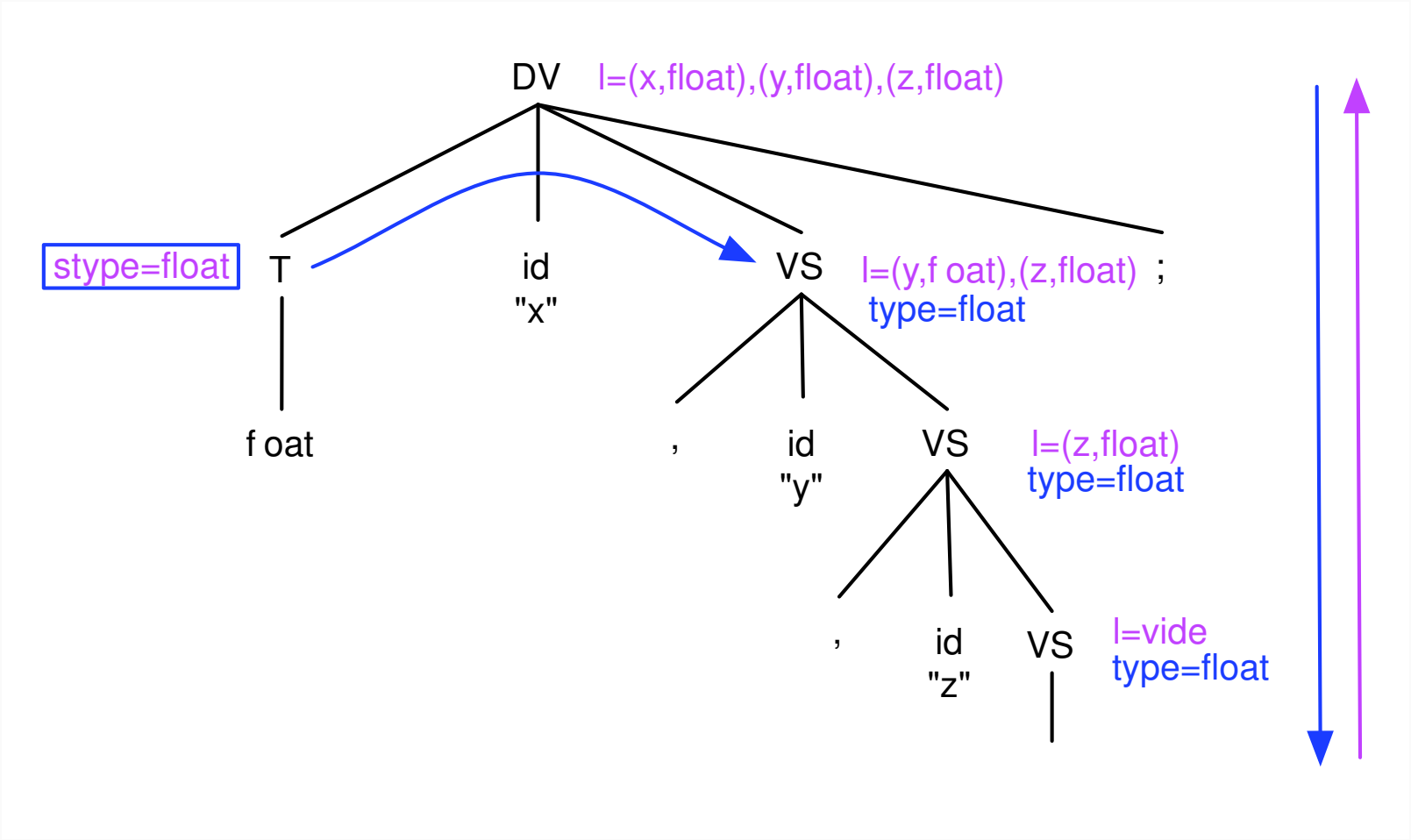
- Soit la grammaire représentant la déclaration de variables :
  1. $DV \rightarrow T\ id\ VS\ ;$
  2. $VS \rightarrow ,\ id\ VS$
  3. $VS \rightarrow \Lambda$
  4. $T \rightarrow int$
  5. $T \rightarrow float$
- Traitement sémantique souhaité : transformer l'entrée "float x,y,z ;" (conforme à la grammaire) en la liste de couple (nom variable, type) "(x :float)(y :float)(z :float)".

Exemple introductif (2)

- Grammaire attribuée :
  - Attributs sémantiques :
    - Pour VS : le type des éléments de la liste (type)
    - Pour T : le type des éléments de la liste (stype)
    - Pour VS et DV : la liste des couples (liste)
  - Actions sémantiques :
    1. $DV \rightarrow T\ id\ VS$;
       $\{DV.liste = (id.txt, T.stype) :: VS.liste\}$
       $\{VS.type = T.stype\}$
    2. $VS \rightarrow ,\ id\ VS_1$
       $\{VS.liste = (id.txt, VS.type) :: VS_1.liste\}$
       $\{VS_1.type = VS.type\}$
    3. $VS \rightarrow \quad \{VS.liste = []\}$
    4. $T \rightarrow int \quad \{T.stype = int\}$
    5. $T \rightarrow float \quad \{T.stype = float\}$
  - Pour utiliser VS.type, il faut qu'il ait été mis à jour depuis T.

une action sémantique ne peut manipuler que les attributs des symboles de la règle de production.

Exemple introductif (3)



Définitions (1)

Attribut sémantique

Un attribut sémantique est une information attachée à un symbole de la grammaire.

Exemple : "liste" est un attribut sémantique de VS et DV.

Attribut hérité

Un attribut est hérité s'il descend dans l'arbre (généralement calculé d'après les attributs du non-terminal père et / ou frères).
L'attribut est affecté quand le non-terminal est en partie droite des règles : $A \rightarrow X_1 \ ... \ X_n \quad \{X_i.h \leftarrow ...\}$.

Exemple : "type" pour VS.

Définitions (2)

Attribut synthétisé

Un attribut est synthétisé s'il remonte dans l'arbre (généralement calculé d'après les attributs des non-terminaux fils).
L'attribut est affecté quand le non-terminal est en partie gauche des règles : $A \rightarrow X_1 \ ... \ X_n \quad \{A.s \leftarrow ...\}$.

Exemple : "stype" pour T et "liste" pour VS et DV.

Synthétisé oux hérité

Par convention, un attribut est synthétisé **oux** hérité, car le principe initial est purement fonctionnel sans effet de bord.

Terminaux

Les terminaux ne possèdent que des attributs synthétisés, car destinés à remonter dans l'arbre. Ils sont généralement calculés par l'analyseur lexical.

Exemple : "txt" pour id.

Définitions (3)

Action sémantique

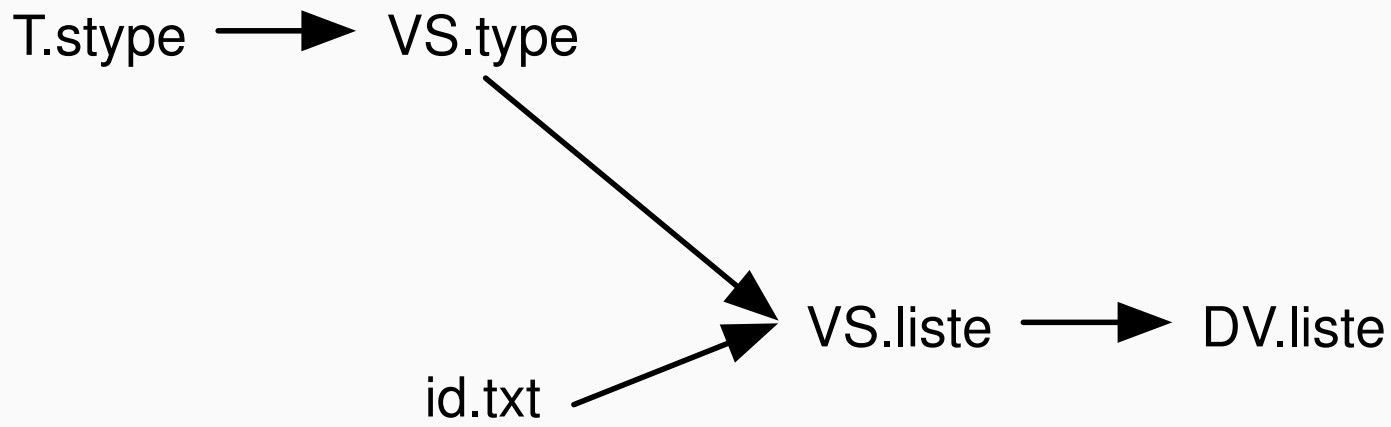
Une action sémantique est du code attaché à une règle de production manipulant et / ou mettant à jour les attributs sémantiques des symboles apparaissant dans la règle de production. On parle aussi de règle sémantique.

Grammaire attribuée

Grammaire + attributs sémantiques + actions sémantiques

Évaluation des grammaires attribuées

En général, la valeur d'un attribut dépend des valeurs des autres attributs. Dans notre exemple le graphe de dépendance est :



Pour qu'une grammaire attribuée soit évaluable il faut qu'il existe un ordre de calcul sans cycle (graphe de dépendance acyclique : DAG).

Parcours d'arbre

a) Choisir arbitrairement le parcours de l'arbre
   - On suppose l'arbre de dérivation construit en mémoire
   - Définir un ordre sur le calcul des attributs d'après les dépendances.
   - Problème : il peut y avoir des cycles.

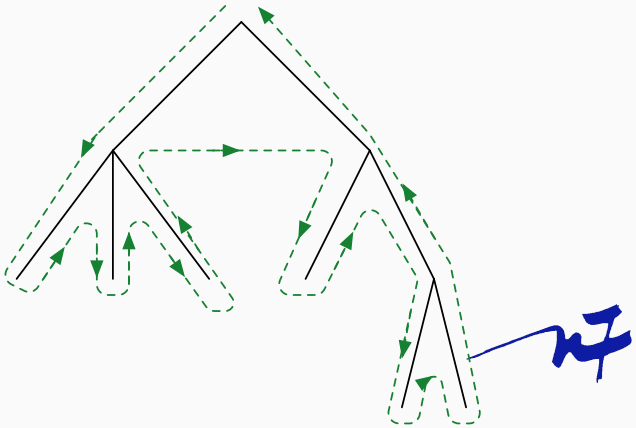
b) Parcours guidé par l'analyse syntaxique ascendante
   - Adapté pour les attributs synthétisés
   - Pas d'évaluation des attributs hérités (ou avec des var. globales)

c) Parcours guidé par l'analyse syntaxique descendante

   $A \rightarrow X_1 X_2 X_3$

   Pour reconnaître A, on étudie X_1, X_2, puis X_3.

- Descente ou frère : attributs hérités
- Remontée : attributs synthétisés
- Problème : si des attributs de X2 dépendent de ceux de X3



Attributs et actions dans Menhir

- Menhir : analyseur ascendant $\Rightarrow$ que des attributs synthétisés
- Un attribut synthétisé pour chaque non-terminal (le nom de l'attribut est local à chaque règle)
  Syntaxe : <nom_att> = <non-terminal>
- Une action sémantique par règle de production, exécutée à la fin de l'analyse de la règle
  Syntaxe : règle { action sémantique }
- Possibilité d'avoir des attributs sur les terminaux, positionnés par l'analyseur lexical
  Syntaxe : %token <type de l'att> <terminal>

Exemple Menhir ♪♪♪

```
%token <int> INT     (* attribut de type int pour le terminal INT *)
%token <string> NAME (* attribut de type string pour le terminal NAME *)
%token XXXX
%token EOF

(* Types des attributs synthétisés des non-terminaux *)
%type <(int*string*int*int)> ligne
(* Axiome et type de l'attribut synthétisé de l'axiome *)
%start <(int*string*int*int) list> main

%%
main:
| stat = ligne EOF      {[stat]}
| stat = ligne m = main {stat::m}
(* Explication de la ligne précédente :
   stat : nom de l'attribut synthétisé du non-terminal "ligne".
   m : nom de l'attribut synthétisé du non-terminal "main".
   {[stat]} et {stat::m} : actions sémantiques permettant de calculer
         la valeur de l'attribut synthétisé du "main" parent. *)

ligne :
| sexe = INT prenom = NAME annee = INT nb = INT {(sexe,prenom,annee,nb)}
| sexe = INT prenom = NAME        XXXX nb = INT {(sexe,prenom,-1,nb)}
```

Exemple Menhir (avec EBNF)

```
%token <int> INT     (* attribut de type int pour le terminal INT *)
%token <string> NAME (* attribut de type string pour le terminal NAME *)
%token XXXX
%token EOF

(* Types des attributs synthétisés des non-terminaux *)
%type <(int*string*int*int)> ligne
(* Axiome et type de l'attribut synthétisé de l'axiome *)
%start <(int*string*int*int) list> main

%%
main: stats = ligne+ EOF {stats}
(* Explication de la ligne précédente :
   stats : nom de l'attribut synthétisé par la répétition du
       non-terminal "ligne" (une liste).
   {stats} : action sémantique permettant de calculer la valeur de
       l'attribut synthétisé du "main" parent : simple propagation. *)

ligne :
| sexe = INT prenom = NAME annee = INT nb = INT {(sexe,prenom,annee,nb)}
| sexe = INT prenom = NAME XXXX nb = INT        {(sexe,prenom,-1,nb)}
```

Bilan

Bilan

1. La phase d'analyse lexicale permet de découper le programme source en tokens qui correspondent aux terminaux de la grammaire.
2. La phase d'analyse syntaxique permet alors de vérifier que les terminaux sont "dans le bon ordre", soit ordonnés de façon conforme à la grammaire.
3. L'ajout d'attributs et d'actions sémantiques à la grammaire permettent de réaliser, lors de l'analyse syntaxique, des traitements sur le code source.
    - Tout le compilateur pourrait être réalisé à l'aide des attributs et actions sémantiques ;
    - mais, généralement, ces attributs et actions sont utilisés pour générer un *arbre syntaxique abstrait* sur lequel les traitement liés à la compilation sont réalisés.

Cinquième partie V

Arbre syntaxique abstrait (AST)

Arbre de dérivation vs Arbre abstrait ♪♪♪

Arbre de dérivation

- 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.

Arbre abstrait (AST : Abstract Syntax Tree)

- 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

Structure des arbres

- L'arbre de dérivation est unique pour une entrée donnée et un analyseur syntaxique, alors que l'AST est indépendant de l'analyse syntaxique.
- 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.

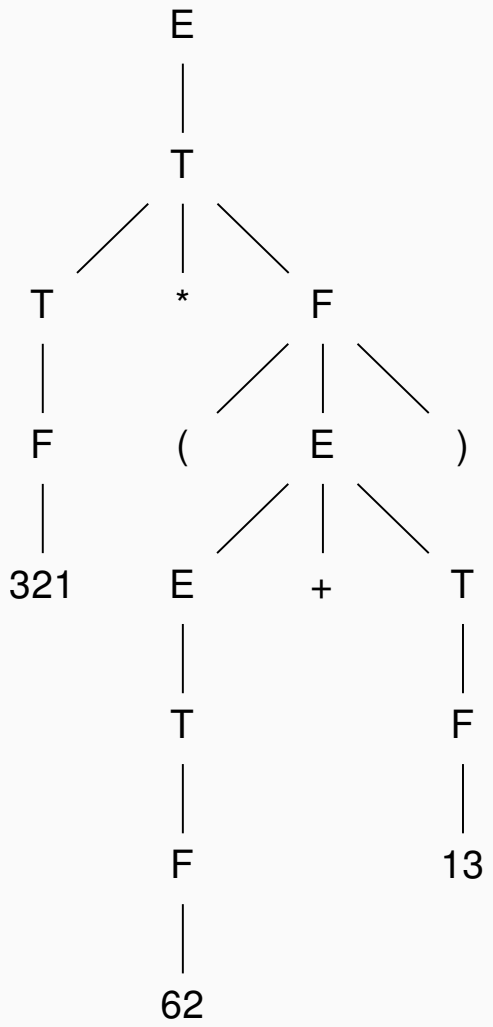
Exemple : Arbre de dérivation vs Arbre abstrait

Source : $321 * (62 + 13)$

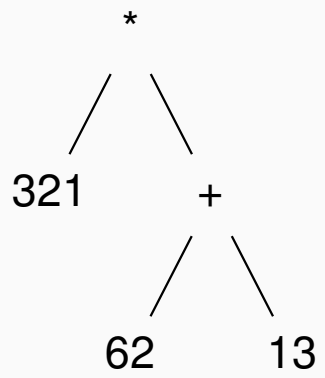
Grammaire des expressions arithmétiques

$E \rightarrow E + T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow F$
$F \rightarrow \textit{entier}$
$F \rightarrow (E)$

Arbre de dérivation



Arbre abstrait



Construction de l'arbre abstrait

- L'arbre abstrait est construit lors de l'analyse syntaxique à l'aide d'attributs et d'actions sémantiques.
- Une fois l'arbre abstrait construit, on pourra le parcourir et le transformer autant de fois que nécessaire pour réaliser le traitement sémantique souhaité.

Exemple

$$
\begin{array}{ll}
E \rightarrow E_1 \ + \ T & \{E.ast = Plus(E_1.ast, T.ast)\} \\
E \rightarrow T & \{E.ast = T.ast\} \\
T \rightarrow T_1 \ * \ F & \{T.ast = Mult(T_1.ast, F.ast\} \\
T \rightarrow F & \{T.ast = F.ast\} \\
F \rightarrow entier & \{F.ast = Entier(entier.val)\} \\
F \rightarrow (\ E \ ) & \{F.ast = E.ast\}
\end{array}
$$

Bilan

Bilan

1. La phase d'analyse lexicale permet de découper le programme source en tokens qui correspondent aux terminaux de la grammaire.
2. La phase d'analyse syntaxique permet alors de vérifier que les terminaux sont "dans le bon ordre", soit ordonnés de façon conforme à la grammaire.
3. L'ajout d'attributs et d'actions sémantiques à la grammaire permet de construire l'arbre abstrait lors de l'analyse syntaxique.
4. L'arbre abstrait constitue une interface naturelle pour l'analyse sémantique. Il sera parcouru et transformé autant de fois que nécessaire pour réaliser le traitement sémantique souhaité.

Sixième partie VI

Analyse sémantique : Application à la compilation

Principe général

La traduction d'un code source en code cible est réalisé par une suite de transformations d'arbres annotés :

- On part de l'arbre syntaxique abstrait.
- Chaque passe transforme un arbre en entrée en un nouvel arbre en sortie, en ajoutant des annotations aux nœuds.
- Une passe peut changer la structure de l'arbre.
- Une passe utilise les annotations précédemment ajoutées pour faire sa transformation.

Notre compilateur simplifié a quatre passes :

- Gestion des identifiants : reconnaître les identifiants (variables, fonctions. . .) et faire le lien entre leurs multiples utilisations ;
- Typage : associer les informations de types aux identificateurs et vérifier la correction du typage ;
- Placement mémoire : déterminer où sont rangés en mémoire les identificateurs et leurs valeurs ;
- Génération de code : production du code assembleur.

Langage RAT

Langage RAT

La grammaire

1. $PROG' \rightarrow PROG\ \$$
2. $PROG \rightarrow FUN\ PROG$
3. $FUN \rightarrow TYPE\ id\ (\ DP\ )\ BLOC$
4. $PROG \rightarrow id\ BLOC$
5. $BLOC \rightarrow \{\ IS\ \}$
6. $IS \rightarrow I\ IS$
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. $I \rightarrow if\ E\ BLOC\ else\ BLOC$
13. $I \rightarrow while\ E\ BLOC$
14. $I \rightarrow return\ E\ ;$
15. $DP \rightarrow \Lambda$
16. $DP \rightarrow TYPE\ id\ DP2$
17. $DP2 \rightarrow ,\ TYPE\ id\ DP2$
18. $DP2 \rightarrow \Lambda$
19. $TYPE \rightarrow bool$
20. $TYPE \rightarrow int$
21. $TYPE \rightarrow rat$
22. $E \rightarrow id\ (\ CP\ )$
23. $E \rightarrow [\ E\ /\ E\ ]$
24. $E \rightarrow num\ E$
25. $E \rightarrow denom\ E$
26. $E \rightarrow id$
27. $E \rightarrow true$
28. $E \rightarrow false$
29. $E \rightarrow entier$
30. $E \rightarrow (\ E\ +\ E\ )$
31. $E \rightarrow (\ E\ *\ E\ )$
32. $E \rightarrow (\ E\ =\ E\ )$
33. $E \rightarrow (\ E\ <\ E\ )$
34. $E \rightarrow (\ E\ )$
35. $CP \rightarrow \Lambda$
36. $CP \rightarrow E\ CP2$
37. $CP2 \rightarrow ,\ E\ CP2$
38. $CP2 \rightarrow \Lambda$

Langage RAT

La grammaire (version EBNF)

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

Rappel : $X*$ est la répétition 0 ou plus de X.

Langage RAT

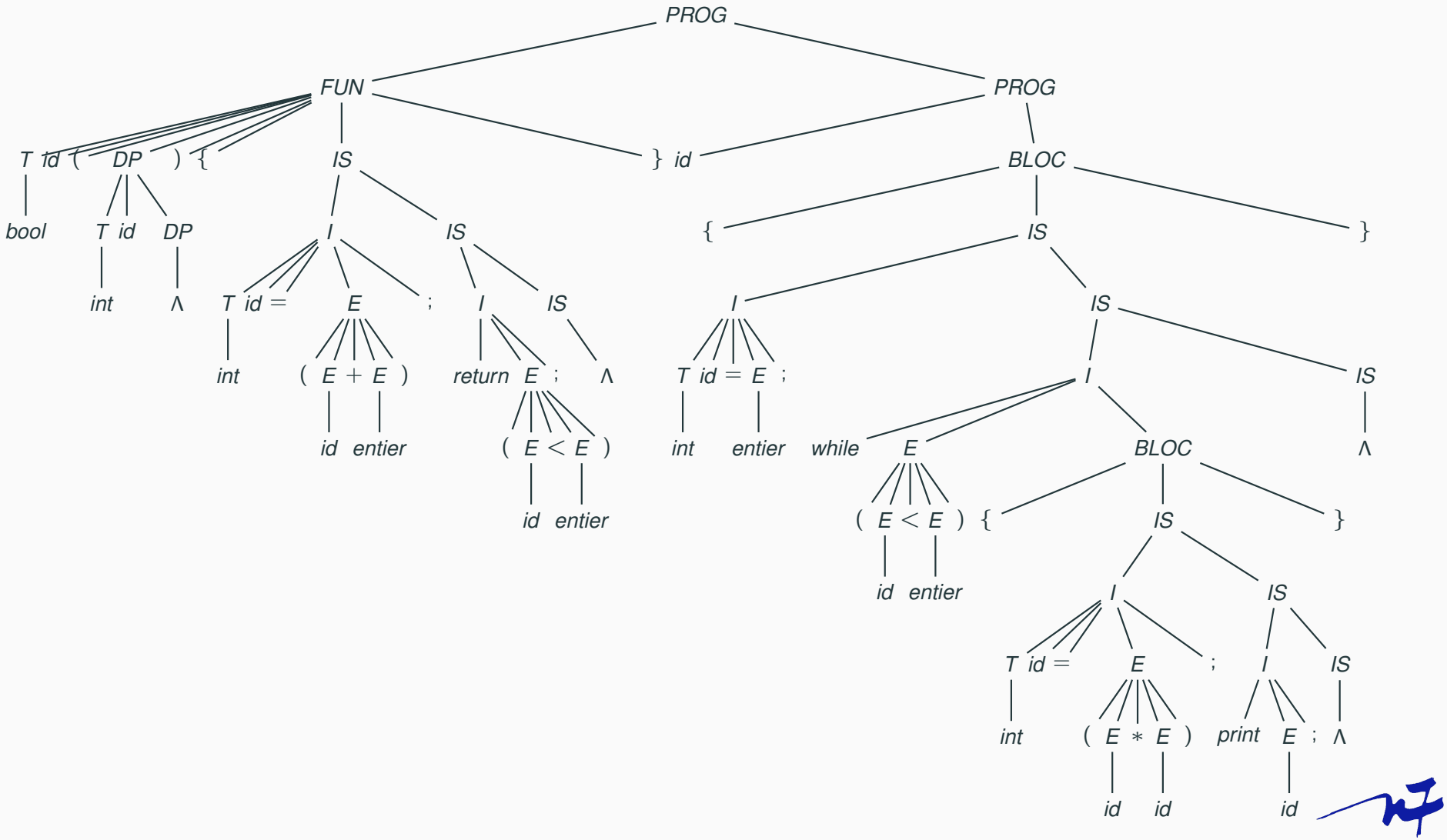
Un exemple

```
// comparaison de deux rationnels
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 (less(a, b)) {print a;} else {print b;}
}
```

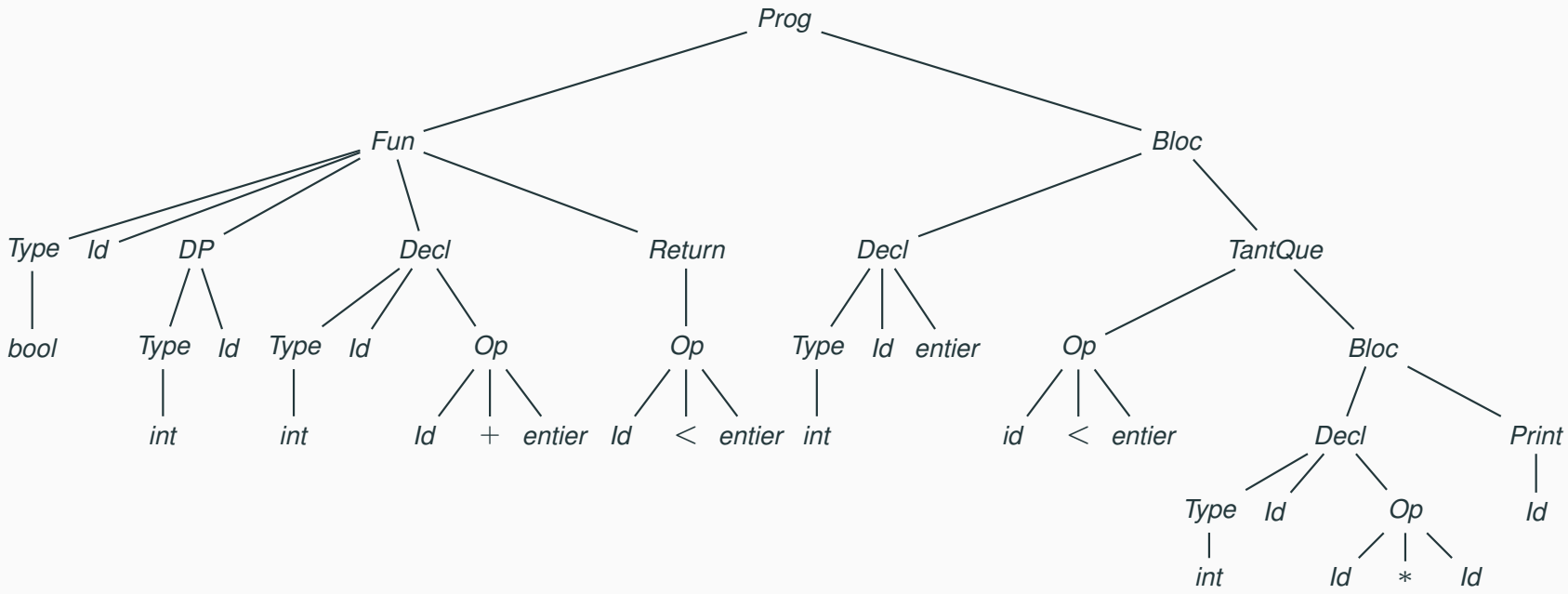
Un exemple simple

```
bool less (int a) {
     int b = (a+3);
     return (b < 6);
}

prog {
   int i = 0;
   while (i<5) {
       int j = (i*i);
       print j;
   }
}
```

Arbre de dérivation



Un arbre abstrait possible



Langage RAT : arbre abstrait

AST vs arbre de dérivation

Par rapport à l'arbre de dérivation :

- Suppression des symboles inutiles : (,(,{,},;...
- Aplatissement des listes
- Ajout de plusieurs cas pour distinguer les instructions et les expressions $\Rightarrow$ permet l'utilisation du pattern matching

Gestion des identifiants

Exemple : introduction de la table des symboles (TDS) ♪♪♪

Code

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

AST

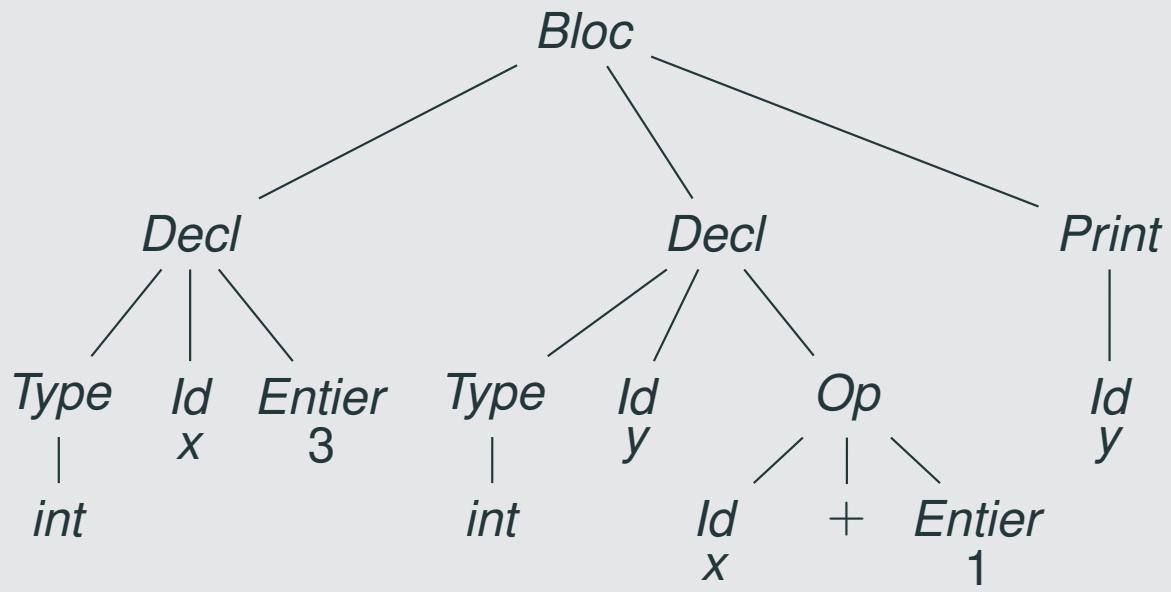


Table des symboles (TDS) : Introduction

Objectifs de la table des symboles

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

Variété des 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	

Table des symboles : spécification

Opérations sur la TDS

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

Exemples d'implantation

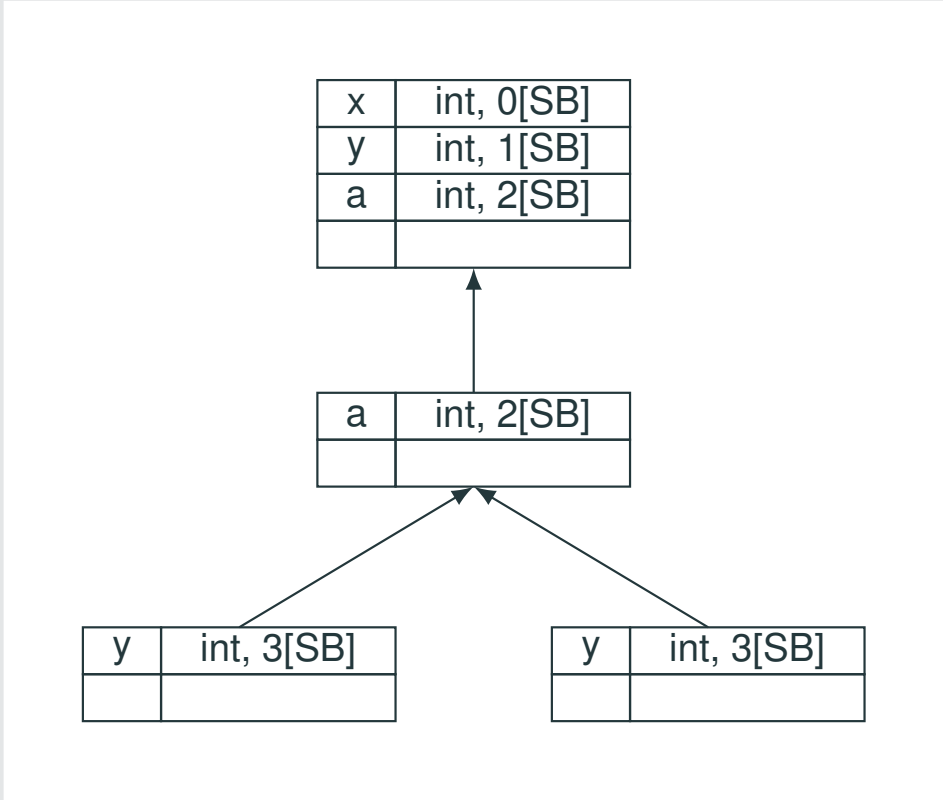
- Tableau
- Liste chaînée
- Arbre binaire de recherche
- Table de hachage
- ...

TDS $\rightarrow$

Nom	Info

Portée des variables

```
main {
int x = 0;
int y = 4;
while (x < y){
  int a = (x + y);
  if (a < 6){
     int y = x;
     print y;
  } else {
     int y = a;
     print y;
  }
}
int a = y;
print a;
}
```



⇒ Modification de la recherche : locale ou globale.

Passe de gestion des identifiants

Les actions sémantiques doivent permettre de :

- propager la table des symboles
- 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. . .)
- créer un nouvel arbre où la TDS sera stockée quand nécessaire (pour qu'elle puisse être utilisée lors d'une nouvelle passe pour un autre traitement)

Pour aller plus loin... ♪♪♪

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

Typage

Typage [3]

Exemple

`"5" + 37` renvoie :

- une erreur à la compilation (OCaml)
- une erreur à l'exécution (Python)
- 42 (Visual Basic ou PHP)
- la chaîne "537" (Java)
- autre chose ?

Objectif

Comment déterminer si une expression est légale et sa sémantique le cas échéant ?

Une (partie d'une) réponse possible est le **typage**.

3. Merci à Jean-Christophe Filliâtre pour l'accès à son cours de Langage de programmation et compilation de l'École Normale Supérieure

Type

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

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 type aux diverses parties d'un programme.

Typage fort vs typage faible

- Typage fort
  - Langages : ADA, OCaml
  - Prémunit contre nombre d'erreurs.
  - Un programme accepté fournit un résultat conforme à son type ou boucle mais n'adopte pas un comportement erratique nuisible.
- Typage "faible"
  - Langages : C
  - Garantie peu claire.
  - Vérification que les données manipulées ont la bonne taille en mémoire.
- Dans le cadre de ce cours : typage fort

Typage statique vs typage dynamique

- Typage dynamique : à l'exécution
  - Langages : Lisp, PHP, Python, Java (liaison dynamique)...
- Typage statique : à la compilation
  - Langages : C, Java, OCaml, ADA...
  - *"Well typed programs do not go wrong"*, Robin Milner
- Dans le cadre de ce cours : typage statique

Inférence de type vs contrôle de type

- Contrôle de types
  - Langages : C, Java, ADA, OCaml
  - Types explicités par le développeur et vérifiés par le compilateur / l'interprête.
- Inférence de type
  - Langages : OCaml
  - Types calculés (et donc vérifiés).
- Dans le cadre de ce cours
  - Cours : inférence et contrôle de type
  - TD et TP : contrôle de type

Exemple

```
# let f a b c = if a then b+1 else c+1;;
val f : bool -> int -> int -> int = <fun>
```

Illustration

- Définition du système d'inférence d'une sous-partie d'OCaml.
- Grammaire de Mini-ML

Expr	→	*Ident*
	|	*Const*
	|	*Expr Binaire Expr*
	|	*Unaire Expr*
	|	`(` *Expr* `)`
	|	`if` *Expr* `then` *Expr* `else` *Expr*
	|	`let` *Ident* `=` *Expr* `in` *Expr*
	|	`fun` *Ident* `->` *Expr*
	|	`(`*Expr* `(`*Expr*`))`
	|	`let rec` *Ident* `=` *Expr* `in` *Expr*
Const	→	*entier* | *booleen*
Unaire	→	`-` | `!`
Binaire	→	`+` | `-` | `*` | `/` | `%` | `&` | `|`
	|	`==` | `!=` | `<` | `<=` | `>` | `>=`

Jugement de typage

- Un jugement de typage s'écrit sous la forme $\sigma \vdash e : \tau$
  - Dans l'environnement σ (par exemple σ est une TDS)
  - l'expression e
  - a pour type τ.
- Système de type : ensemble de jugements de typage

Constantes

$$\sigma \vdash \mathit{entier} : \mathit{int} \qquad \sigma \vdash \mathit{true} : \mathit{bool} \qquad \sigma \vdash \mathit{false} : \mathit{bool}$$

Accès à l'environnement

$$\frac{x \in \sigma \qquad \sigma(x) = \tau}{\sigma \vdash x : \tau}$$

Opérateur binaire

$$\frac{\sigma \vdash e_1 : \tau_1 \quad \sigma \vdash e_2 : \tau_2 \quad \tau_1 \times \tau_2 = dom\, op \quad \tau = codom\, op}{\sigma \vdash e_1\ op\ e_2 : \tau}$$

Opérateur unaire

$$\frac{\sigma \vdash e : \tau \qquad \tau = dom\, op \qquad \tau' = codom\, op}{\sigma \vdash op\ e : \tau'}$$

Typage - Jugement de typage - Mini-ML

Conditionnelle ♪♪♪

$$\frac{\sigma \vdash e_1 : \texttt{bool} \qquad \sigma \vdash e_2 : \tau \qquad \sigma \vdash e_3 : \tau}{\sigma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}$$

Définition locale ♪♪♪

$$\frac{\sigma \vdash e_1 : \tau_1 \qquad (x, \tau_1) :: \sigma \vdash e_2 : \tau_2}{\sigma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

Typage - Jugement de typage - Mini-ML

Définition de fonction ♪♪♪

$$\frac{(x, \tau_1) :: \sigma \vdash e : \tau_2}{\sigma \vdash \texttt{fun}\ x\ \texttt{->}\ e : \tau_1 \to \tau_2}$$

Appel de fonction ♪♪♪

$$\frac{\sigma \vdash e_1 : \tau_1 \to \tau_2 \qquad \sigma \vdash e_2 : \tau_1}{\sigma \vdash (e_1\ (e_2)) : \tau_2}$$

Définition récursive ♪♪♪

$$\frac{(x, \tau_1) :: \sigma \vdash e_1 : \tau_1 \qquad (x, \tau_1) :: \sigma \vdash e_2 : \tau_2}{\sigma \vdash \texttt{let rec}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

Typage - Jugements de typage et inférence de type

Principe de l'inférence

- Les jugements de typage peuvent être utilisés pour construire un algorithme d'inférence.
- Lors de l'écriture de la preuve de typage, des contraintes sur les types apparaissent.
- Ces contraintes permettent de déduire le type des expressions.

Exemple - Définition locale ♪♪♪

$$\dfrac{\dfrac{}{[] \vdash 1 : int} \quad \dfrac{\dfrac{}{[(x, int)] \vdash x : int} \quad \dfrac{}{[(x, int)] \vdash 1 : int} \quad \dfrac{}{int \times int = dom +} \quad \dfrac{}{int = codom +}}{[(x, int)] \vdash x + 1 : int}}{[] \vdash \texttt{let}\ x = 1\ \texttt{in}\ x + 1 : int}$$

Exemple - Définition de fonction et conditionnelle ♪♪♪

$$\dfrac{\dfrac{}{[(a, \mathit{bool}); (b, \tau_6); (c, \tau_6)] \vdash a : \mathit{bool}} \quad \dfrac{}{[(a, \mathit{bool}); (b, \tau_6); (c, \tau_6)] \vdash b : \tau_6} \quad \dfrac{}{[(a, \mathit{bool}); (b, \tau_6); (c, \tau_6)] \vdash c : \tau_6}}{\dfrac{[(a, \mathit{bool}); (b, \tau_6); (c, \tau_6)] \vdash \texttt{if}\ a\ \texttt{then}\ b\ \texttt{else}\ c : \tau_6}{\dfrac{[(a, \mathit{bool}); (b, \tau_6)] \vdash \texttt{fun}\ c\ \texttt{->}\ \texttt{if}\ a\ \texttt{then}\ b\ \texttt{else}\ c : \tau_6 \to \tau_6}{\dfrac{[(a, \mathit{bool})] \vdash \texttt{fun}\ b\ \texttt{->}\ \texttt{fun}\ c\ \texttt{->}\ \texttt{if}\ a\ \texttt{then}\ b\ \texttt{else}\ c : \tau_6 \to \tau_6 \to \tau_6}{[] \vdash \texttt{fun}\ a\ \texttt{->}\ \texttt{fun}\ b\ \texttt{->}\ \texttt{fun}\ c\ \texttt{->}\ \texttt{if}\ a\ \texttt{then}\ b\ \texttt{else}\ c : \mathit{bool} \to \tau_6 \to \tau_6 \to \tau_6}}$$

Typage - Inférence de type

Bilan

- Besoin d'un mécanisme d'unification de types
- Le type inféré est le plus général que puisse prendre une expression
- Code source plus aéré / lisible
- Moyen de vérification
  - si impossible d'inférer le type $\rightarrow$ erreur
  - si le type inféré n'est pas celui espéré $\rightarrow$ alarme

Typage - RAT

Exemple

```
bool foo(int x){
  int y = 1;
  return (x<y);
}

prog{
  int a = 4 ;
  print (foo(a));
}
```

Vérifications

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

Typage - Jugement de typage - RAT

Sémantique naturelle du typage ♪♪♪

- Jugement de typage : $\sigma, \tau_r \vdash e : \tau, \sigma'$
  - σ : l'environnement de typage (TDS)
  - τ_r : optionnel, le type de retour de la fonction en cours d'analyse
  - e : l'élément à typer (expression, instruction...)
  - τ : le type
  - σ' : optionnel, les nouvelles informations à ajouter à l'environnement de typage

♪♪♪

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

Base ♪♪♪

- $$\frac{x \in \sigma \qquad \sigma(x) = \tau}{\sigma \vdash x : \tau}$$

  Si x a été ajouté plusieurs fois à σ, $\sigma(x)$ renvoie le dernier type associé à x.

- $\sigma \vdash \textit{true} : \textit{bool}$
- $\sigma \vdash \textit{false} : \textit{bool}$
- $\sigma \vdash \textit{entier} : \textit{int}$

Typage - Jugement de typage - RAT

Les cas d'erreur ne seront pas donnés.

Expression (1)

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

Expression (2)

- $\dfrac{\sigma \vdash E_1 : int \quad \sigma \vdash E_2 : int}{\sigma \vdash (E_1 + E_2) : int}$
- $\dfrac{\sigma \vdash E_1 : rat \quad \sigma \vdash E_2 : rat}{\sigma \vdash (E_1 + E_2) : rat}$
- $\dfrac{\sigma \vdash E_1 : int \quad \sigma \vdash E_2 : int}{\sigma \vdash (E_1 * E_2) : int}$
- $\dfrac{\sigma \vdash E_1 : rat \quad \sigma \vdash E_2 : rat}{\sigma \vdash (E_1 * E_2) : rat}$
- $\dfrac{\sigma \vdash E_1 : int \quad \sigma \vdash E_2 : int}{\sigma \vdash (E_1 = E_2) : bool}$
- $\dfrac{\sigma \vdash E_1 : bool \quad \sigma \vdash E_2 : bool}{\sigma \vdash (E_1 = E_2) : bool}$
- $\dfrac{\sigma \vdash E_1 : int \quad \sigma \vdash E_2 : int}{\sigma \vdash (E_1 < E_2) : bool}$

On se limitera à ces signatures ♪♪♪.

Typage - Jugement de typage - RAT

Structures de contrôle ♪♪♪

- $$\frac{\sigma \vdash E : bool \qquad \sigma, \tau_r \vdash BLOC_1 : void \qquad \sigma, \tau_r \vdash BLOC_2 : void}{\sigma, \tau_r \vdash \textit{if } E \; BLOC_1 \; \textit{else } BLOC_2 : void, []}$$
- $$\frac{\sigma \vdash E : bool \qquad \sigma, \tau_r \vdash BLOC : void}{\sigma, \tau_r \vdash \textit{while } E \; BLOC : void, []}$$

Typage - Jugement de typage - RAT

Déclaration / affectation ♪♪♪

- $$\frac{\sigma \vdash TYPE : \tau \qquad \sigma \vdash E : \tau}{\sigma, \tau_r \vdash TYPE\ id = E : void, [id, \tau]}$$
- $$\frac{\sigma \vdash id : \tau \qquad \sigma \vdash E : \tau}{\sigma, \tau_r \vdash id = E : void, []}$$

Instructions

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

Typage - Jugement de typage - RAT

Déclaration de fonction ♪♪♪

- $$\frac{A \quad B \quad C}{\sigma \vdash TYPE\ id\ (DP)\ BLOC : void, [id, \tau_p \to \tau_r]}$$
  - A : $\sigma \vdash TYPE : \tau_r$
  - B : $\sigma \vdash DP : \tau_p, \sigma_p \qquad (DP = TYPE_1\ id_1, \ldots TYPE_n\ id_n)$
  - C : $(id, \tau_p \to \tau_r) :: \sigma_p @ \sigma, \tau_r \vdash BLOC : void$
- $$\frac{\sigma \vdash TYPE_1 : \tau_1 \quad \ldots \quad TYPE_n : \tau_n}{\sigma \vdash TYPE_1\ id_1, \ \ldots\ TYPE_n\ id_n : \tau_1 \times \cdots \times \tau_n, [(id_1, \tau_1); \ldots; (id_n, \tau_n)]}$$

Typage - Jugement de typage - RAT

Appel de fonction

- $$\frac{\sigma \vdash id : \tau_1 \to \tau_2 \qquad E_1, \ \ldots \ E_n : \tau_1}{\sigma \vdash id \ (E_1 \ \ldots \ E_n) : \tau_2}$$
- $$\frac{\sigma \vdash E_1 : \tau_1 \quad \ldots \quad E_n : \tau_n}{\sigma \vdash E_1, \ \ldots \ E_n : \tau_1 \times \cdots \times \tau_n}$$

Retour de fonction ♪♪♪

- $$\frac{\sigma \vdash E : \tau_r}{\sigma, \tau_r \vdash return \ E : void, []}$$

Typage - Jugement de typage - RAT

Suite d'instructions ♪♪♪

- $\dfrac{\sigma, \tau_r \vdash IS : void, \sigma'}{\sigma, \tau_r \vdash \{IS\} : void} \qquad \dfrac{\sigma, \tau_r \vdash IS : void, \sigma'}{\sigma, \tau_r \vdash BLOC : void}$

(*BLOC* et $\{IS\}$ sont la même chose)

- $\dfrac{\sigma, \tau_r \vdash I : void, \sigma' \qquad \sigma' @ \sigma, \tau_r \vdash IS : void, \sigma''}{\sigma, \tau_r \vdash I\ IS : void, \sigma'' @ \sigma'}$
- $\sigma, \tau_r \vdash \quad : void$

Le programme ♪♪♪

- $\dfrac{\sigma \vdash FUN : void, \sigma' \qquad \sigma' @ \sigma \vdash PROG : void, \sigma''}{\sigma \vdash FUN\ PROG : void, \sigma'' @ \sigma'}$
- $\dfrac{\sigma, void \vdash BLOC : void}{\sigma \vdash id\ BLOC : void}$

Remarque : l'identifiant du bloc du programme principal n'est pas utilisé.

Typage - Jugements de typage et contrôle de types

Principe du contrôle de type

- Les jugements de typage peuvent être utilisés pour construire un algorithme de contrôle de type.
- Dans le cas du contrôle de type, les types sont connus à priori, ils doivent être vérifiés et non calculés.
- Il n'y a donc pas besoin d'un mécanisme d'unification de type.

Typage de l'exemple

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

A :

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

R :

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

B :

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

Pour aller plus loin...

Réfléchissez aux réponses avant d'écouter les audio qui proposent des pistes de solutions.

- Comment gérer l'héritage de type ?

- Comment construire et nommer un nouveau type ? (par exemple des couples. . .)

Pour aller plus loin...

Réfléchissez aux réponses avant d'écouter les audio qui proposent des pistes de solutions.

- Comment gérer l'héritage de type ?
  - ♪♪♪

    Déclaration :

    $$\frac{\sigma \vdash \mathit{TYPE} : \tau_1 \qquad \sigma \vdash E : \tau_2 \qquad (\mathit{estCompatible}\ \tau_1\ \tau_2)}{\sigma \vdash \mathit{TYPE}\ \mathit{id} = E : \mathit{void}, \{\mathit{id} : \tau_1\}}$$
- Comment construire et nommer un nouveau type ? (par exemple des couples. . .)

Pour aller plus loin...

Réfléchissez aux réponses avant d'écouter les audio qui proposent des pistes de solutions.

- Comment gérer l'héritage de type ?
  - ♪♪♪

    Déclaration :

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

- Comment construire et nommer un nouveau type ? (par exemple des couples...)
  - ♪♪♪

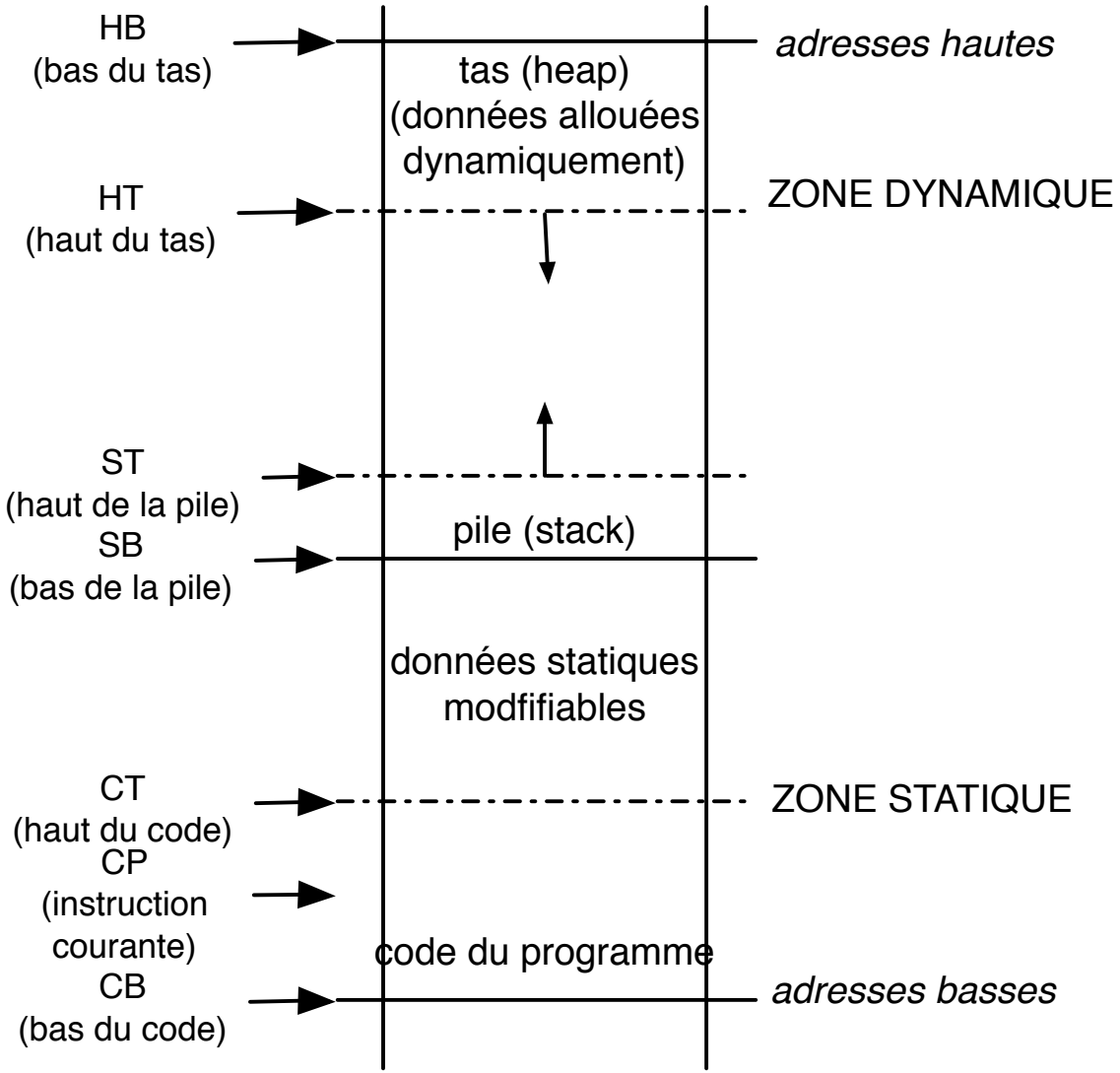
Placement mémoire

Placement mémoire

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, zone définie par le compilateur 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.

Placement mémoire



Variables locales

Où placer les variables locales d'une fonction / du programme principal ?
Où placer les paramètres d'appel d'une fonction ?

- Une zone statique réservée, déterminée à la compilation
  $\rightarrow$ pas de récursivité
- Dans la pile d'appel
  $\rightarrow$ placement relatif à l'instance courante de la fonction

Placement 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

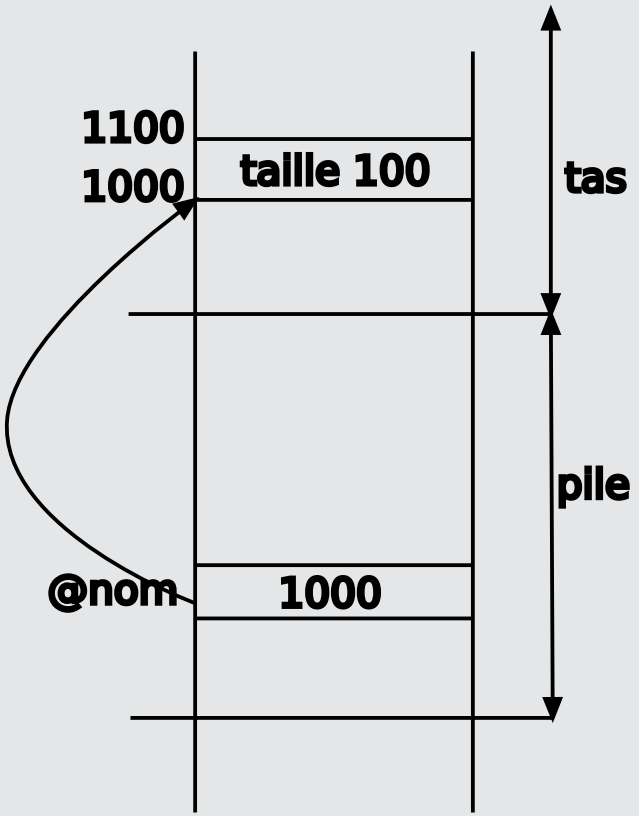
Placement mémoire

Allocation dynamique

- Pointeur

```
foo(){
    char *nom;
    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).

Placement mémoire

Libération de la mémoire

- 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)
  Problème : risque d'erreur (libération à tort) ou d'oubli (fuite mémoire)

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
...		

La pile contient :

- les informations pour l'appel de fonction (à suivre)
- les valeurs d'un calcul intermédiaire
- les variables du programme principal
- les paramètres et les variables locales des fonctions

Placement mémoire

Variables

- Le compilateur a besoin de calculer leur adresse (déplacement par rapport à la base de la pile) et leur 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;
}
```

Quelles sont les adresses des variables ?

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;}
```

9		
8		
7		
6		
5		
4		
3		
2		
1		
0		← SB / ST

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;}
```

9		
8		
7		
6		
5		
4		
3		
2		← ST
1	1	
0	1	← SB @a=0[SB]

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;}
```

9		
8		
7		
6		
5		
4		← ST
3	2	
2	1	@b=2[SB]
1	1	
0	1	← SB @a=0[SB]

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;}
```

9		
8		
7		
6		
5		← ST
4	2	@c=4[SB]
3	2	
2	1	@b=2[SB]
1	1	
0	1	← SB @a=0[SB]

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;}
```

9			
8			
7			
6			
5		← ST	
4	2		@c=4[SB]
3	2		
2	1		@b=2[SB]
1	1		
0	1	← SB	@a=0[SB]

- *num a* $<$ 2 $\rightarrow$ else
- Calculs effectués dans la pile
- Pile nettoyée après calcul

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;}
```

9			
8			
7			
6		← ST	
5	0		@i=5[SB]
4	2		@c=4[SB]
3	2		
2	1		@b=2[SB]
1	1		
0	1	← SB	@a=0[SB]

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;}
```

9			
8			
7			
6		← ST	
5	0		@i=5[SB]
4	2		@c=4[SB]
3	2		
2	1		@b=2[SB]
1	1		
0	1	← SB	@a=0[SB]

- *i* n'existe plus après la conditionnelle
- La pile doit être nettoyée

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;}
```

9			
8			
7			
6			
5		← ST	@i=5[SB]
4	2		@c=4[SB]
3	2		
2	1		@b=2[SB]
1	1		
0	1	← SB	@a=0[SB]

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;}
```

9			
8			
7		← ST	
6	3		
5	3		@i=5[SB]=@d
4	2		@c=4[SB]
3	2		
2	1		@b=2[SB]
1	1		
0	1	← SB	@a=0[SB]

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;}
```

9			
8		← ST	
7	4		@e=7[SB]
6	3		
5	3		@i=5[SB]=@d
4	2		@c=4[SB]
3	2		
2	1		@b=2[SB]
1	1		
0	1	← SB	@a=0[SB]

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;}
```

9			
8		← ST	
7	4		@e=7[SB]
6	3		
5	3		@i=5[SB]=@d
4	2		@c=4[SB]
3	2		
2	1		@b=2[SB]
1	1		
0	1	← SB	@a=0[SB]

Adresse de *a*1 ?

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;}
```

9			
8		← ST	
7	4		@e=7[SB]
6	3		
5	3		@i=5[SB]=@d
4	2		@c=4[SB]
3	2		
2	1		@b=2[SB]
1	1		
0	1	← SB	@a=0[SB]

Adresse de $a1$? $\rightarrow$ 5[SB]

Fonctions

- Le compilateur a besoin de calculer les adresses des paramètres et des variables locales des fonctions.
- Exemple

```
int plus1 (int a, int b){
   int c = 1;
   return (a+(b+c));
}
int foo(int d){
    return plus1(d, 3);
}
fonction{
    int x = 3;
    int y = 4;
    print plus1((x+1), y);
    int i = 1;
    int j = 2;
    print plus1(i, j);
    print foo(i);
}
```

⇒ Quelles sont les adresses de `a`, `b` et `c` ?

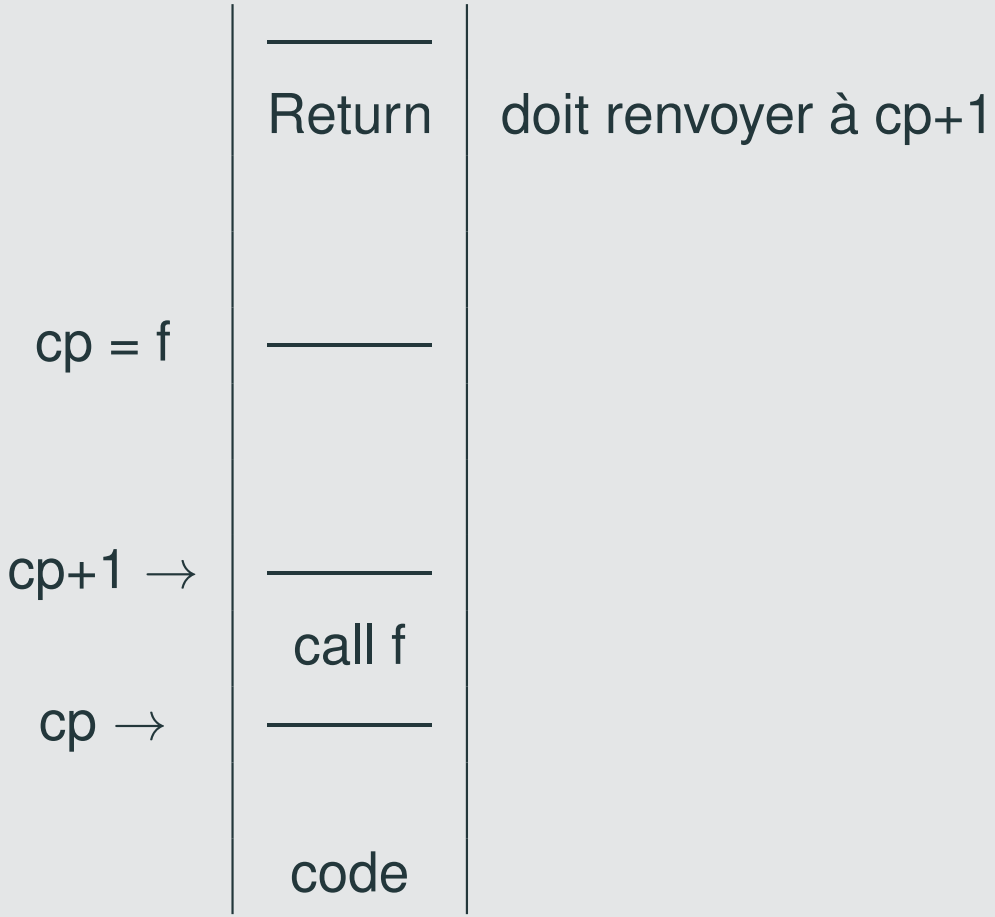
Enregistrement d'activation

- 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
- 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 faut sauvegarder l'adresse de l'instruction suivant l'appel.
    - l'état de la pile : on a besoin d'un lien dynamique «représentant» l'appelant.

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

Placement mémoire

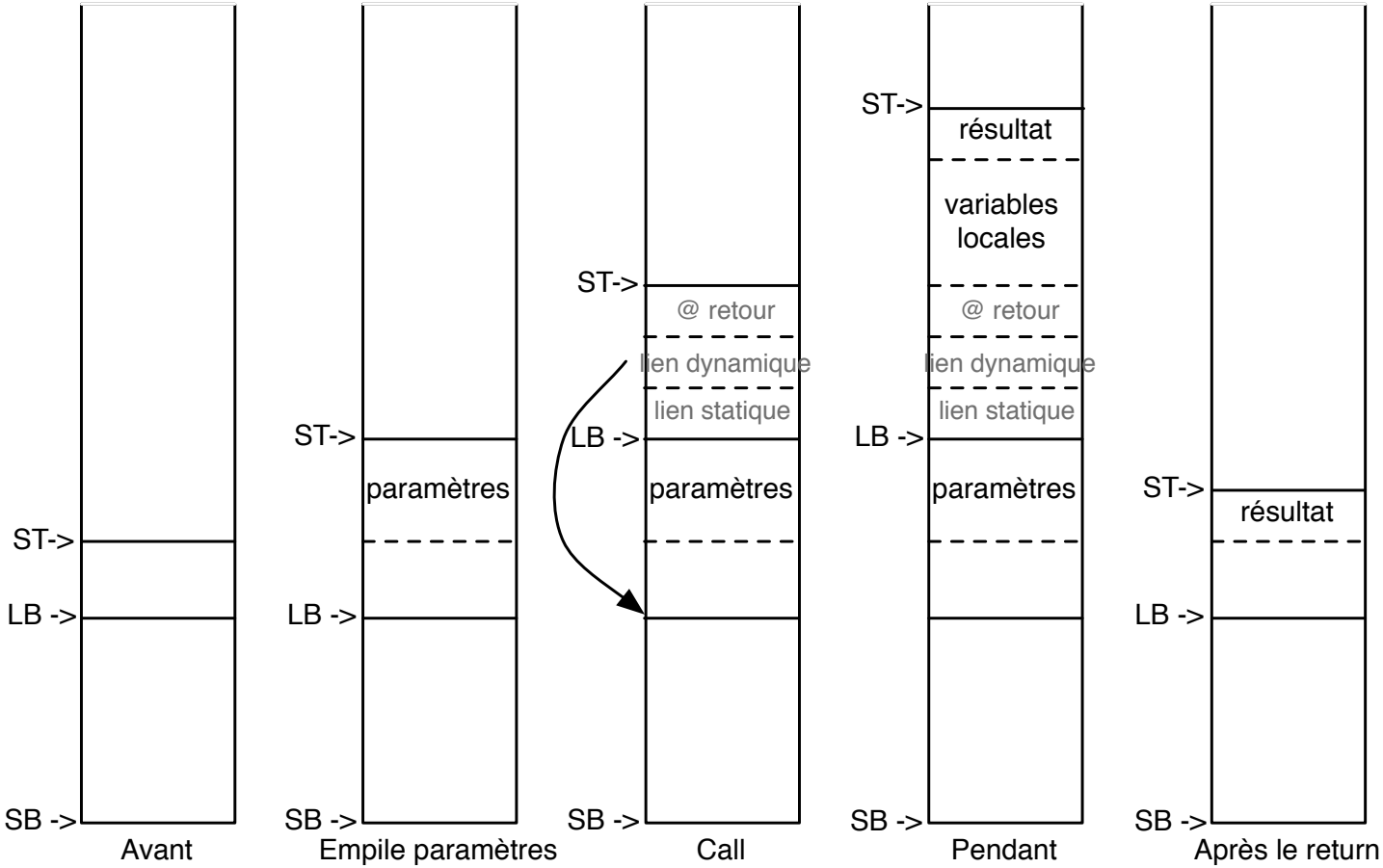
Enregistrement d'activation : l'adresse de retour



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 permet de remonter la pile des appels.



ST = stack top ; LB = link base ; SB = stack base

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 de fonctions 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.

Placement mémoire

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.

Placement mémoire

Retour sur l'exemple (transparent 132)

- x : 0[SB]
- y : 1[SB]
- a : -2[LB]
- b : -1[LB]
  $\Rightarrow$ Attention à l'ordre des paramètres !
- c : 3[LB] (taille de l'enregistrement d'activation)
- i : 2[SB]
- j : 3[SB]
- d : -1[LB]

Placement mémoire

Pour aller plus loin...

- Pas nécessairement besoin de pile des variables locales pour faire l'appel de fonction si pas de récursivité.
- La pile des variables locales pourrait être distincte de la pile des enregistrements d'activation (mais ce n'est jamais le cas)

Génération de code

Machine TAM

La machine TAM est une machine à pile, sans registre de données. Les instructions agissent sur la pile et les registres processeurs (CP - Code Pointer, SB/ST - Stack Base / Top, LB - Link Base, HB/HT - Heap Base / Top).

Principe

Une instruction consomme entre 0 et 3 mots en sommet de pile (elle les dépile), modifie certains registres (CP toujours, LB parfois, ST indirectement), et empile éventuellement un résultat.

Instructions

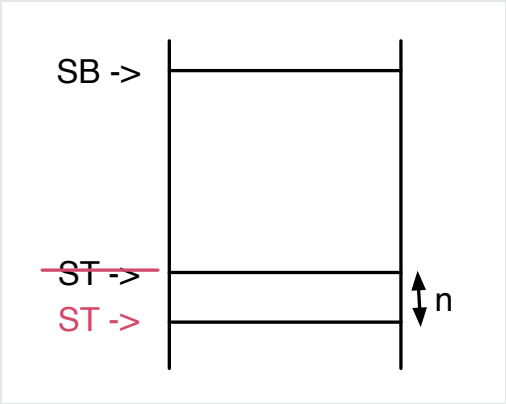
- Flot de contrôle : JUMP, JUMPIF, HALT, CALL, RETURN
- Pile : PUSH, POP, LOADL, LOADA, LOAD, LOADI, STORE, STOREI
- Opérations arithmétiques et logiques, allocation mémoire : SUBR

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

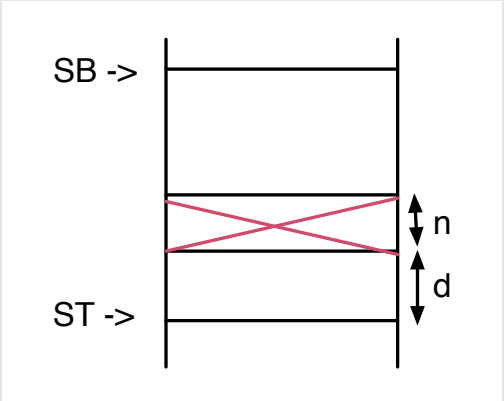
- etiq : déclaration d'une étiquette
- JUMP etiq : déplace la position de CP pour pointer sur l'adresse mise à la place de l'étiquette au moment de l'assemblage
- JUMPIF (n) etiq : vérifie si la donnée en tête de pile est *n* et si c'est le cas déplace la position de CP pour pointer sur l'adresse mise 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 l'opération prédéfinie *op*, consommation des arguments en sommet de pile.
- HALT : arrêt de la machine TAM.

TAM : Instructions agissant sur la pile

- PUSH n : agrandit la pile de *n* mots



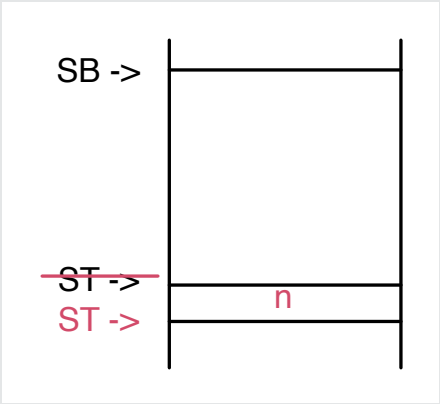
- POP (d) n : garde les *d* premiers et supprime les *n* suivants



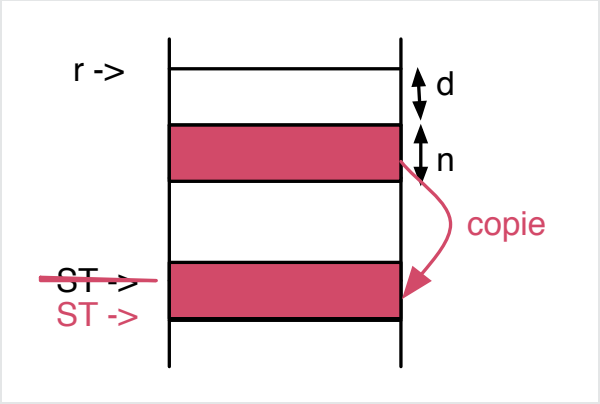
(Attention, les dessins sont faits avec une pile qui croît de haut en bas, contrairement à ceux sur le placement mémoire)

TAM : Instructions agissant sur la pile

- LOADL n : Empile la valeur *n*

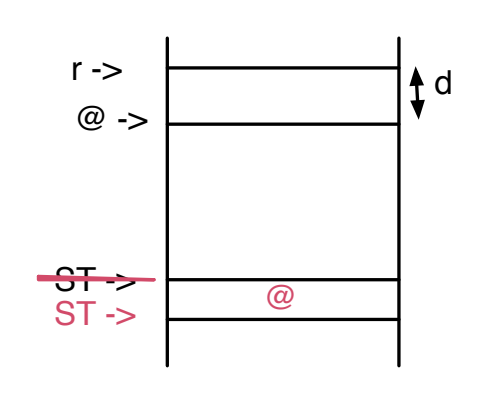


- LOAD (n) d[r] : copie en sommet de pile le bloc de taille *n* qui est à l'adresse *d*[*r*]

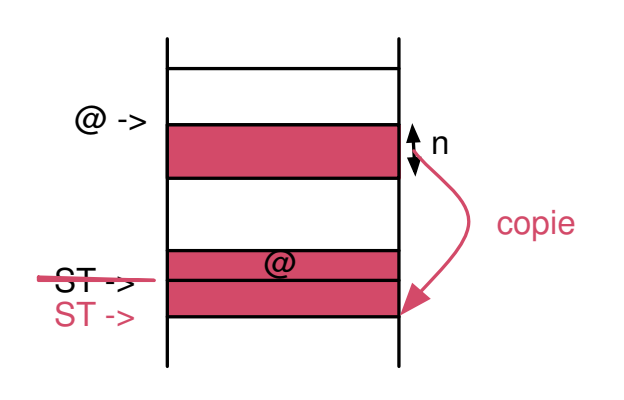


TAM : Instructions agissant sur la pile

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

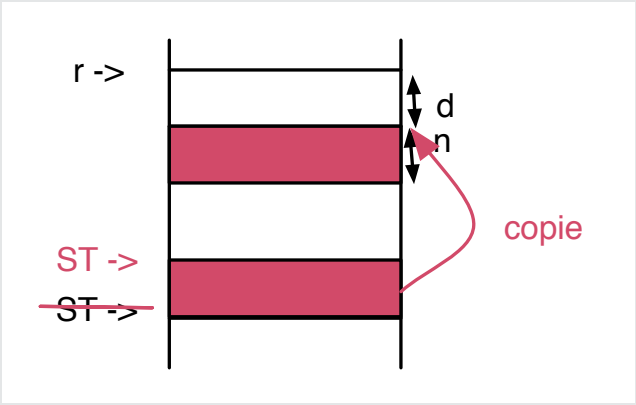


- LOADI (n) : prend l'adresse en sommet de pile, et copie en sommet de pile le bloc de taille n de cette adresse

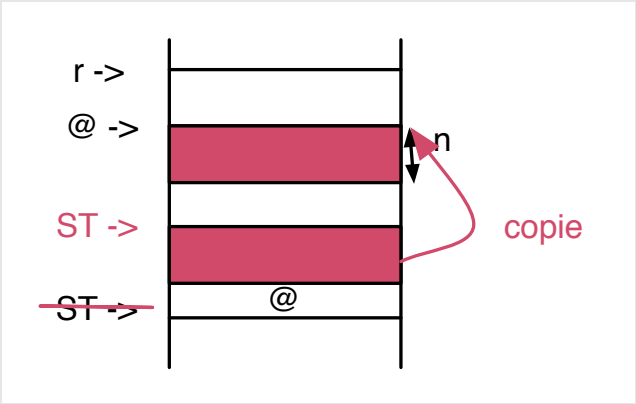


TAM : Instructions agissant sur la pile

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

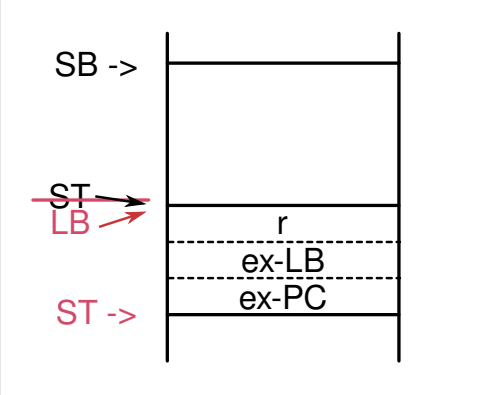


- STOREI (n) : prend l'adresse en sommet de pile, et déplace le bloc de taille *n* en sommet de pile à l'adresse.



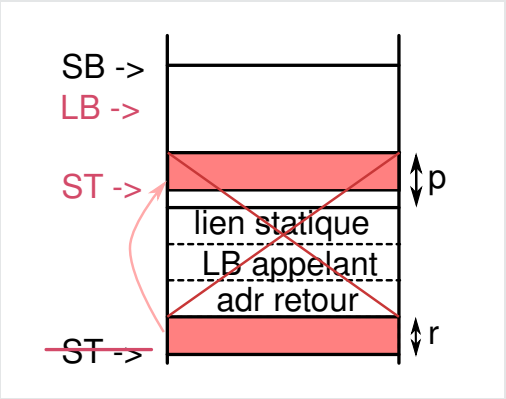
TAM : gestion des fonctions

- CALL (r) op : appel de la procédure *op*. L'enregistrement d'activation contient 3 champs : le lien dynamique (base d'enregistrement d'activation de l'appelant), le lien statique (pour définition imbriquée – passé en argument *r* – nous ne l'utilisons pas), l'adresse de l'instruction à exécuter une fois l'appel fini. Le registre LB contient la base de l'enregistrement d'activation (le sommet de pile au moment de l'appel).



TAM : gestion des fonctions

- RETURN (r) p : fin d'un appel de fonction : *r* est la taille du résultat et *p* la taille des paramètres. Enlève de la pile un bloc de taille *p* à profondeur *r* et laisse un bloc de taille *r* avec les résultats ; restaure LB et CP à partir de l'enregistrement d'activation ; positionne ST à l'ancien LB - *p* + *r*.



RAT $\rightarrow$ 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 plus1(x, y);
}
```

RAT → TAM : Exemple simple avec appel de fonction.

```
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 TAM

RAT $\rightarrow$ TAM : Exemple simple avec appel de fonction.

```
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
RETURN (1) 2        ;fin de la fonction
HALT                ;force l'arrêt si pas de RETURN
```

Dans l'exemple, le HALT en fin du code de la fonction est inutile, car il est certain qu'il y a un RETURN.

Démonstration de Itam.

♪♪♪

Cas particulier de TAM

- Cas particulier d'une machine à pile
- Pas de registres généraux

Pour un assembleur plus évolué ?

- x86-64 [4]
- On traitera sommairement la phase d'analyse qui alloue une variable à un registre

4. Source : cours de compilation de l'ENS - Jean-Christophe Filliâtre

L'architecture x86-64

- 64 bits
  - Opérations arithmétiques, logiques et de transfert de/vers la mémoire se font avec des entiers représentés sur 64 bits.
  - Taille d'un pointeur 64 bits
- l'accès à la mémoire est coûteux $\rightarrow$ registres processeur

Les registres de l'architecture x86-64

64 bits	32 bits	16 bits (15:0)	8 bits hauts (15:8)	8 bits bas (7:0)
rax	eax	ax	ah	al
rbx	ebx	bx	bh	bl
rcx	ecx	cx	ch	cl
rdx	edx	dx	dh	dl
rsi	esi	si		sil
rdi	edi	di		dil
rbp	ebp	bp		bpl
rsp	esp	sp		spl
r8	r8d	r8w		r8b
r9	r9d	r9w		r9b
r10	r10d	r10w		r10b
r11	r11d	r11w		r11b
r12	r12d	r12w		r12b
r13	r13d	r13w		r13b
r14	r14d	r14w		r14b
r15	r15d	r15w		r15b

Exemple : "hello, word"

```
    .text                   # indique que ce qui suit est un programme
    .globl  main            # rend le symbole main visible, début exécutable
main:
    movq    $message, %rdi  # place l'adresse où est stocké le message dans rdi
    call    puts            # affiche sur la sortie standard le contenu de rdi
    movq    $0, %rax        # place 0 dans rax (valeur de retour du programme)
    ret                     # termine le programme
    .data                   # signifie que ce qui suit est des données
message:                    # adresse à laquelle la chaîne est stockée
    .string "hello, world"
```

$ signifie qu'on veut la valeur et non l'adresse.

Exécution

```
> clang hello.s -o hello
> ./hello
hello, world
```

Génération de code x86-64

Instructions

- Des milliers...
- Opérandes d'une instructions :
  - opérande immédiate : $\$n$ (entier n sur 32 bits)
  - registre
  - emplacement mémoire
    - adresse immédiate (entier 32 bits)
    - adresse indirecte : (r) (adresse contenue dans le registre r)
    - forme générale : $n(r_1, r_2, m)$ ($n + r_1 + m * r_2$ où $m = 1, 2, 4, 8$)
  - pas plus d'un accès mémoire par instruction

Transfert de données

- `mov` op_1 op_2 : copie de op_1 dans op_2
  - Si taille des données pas déterminable :
  - `movb` (1 octet), `movw` (2 octets), `movl` (4 octets), `movq` (8 octets)
  - `movq $42, (%rdi)` : écrit l'entier 42 sur 64 bits à l'adresse contenue dans `%rdi`.

Opérations arithmétiques et logiques

- Opérations arithmétiques binaires
  - `add` (addition), `sub` (soustraction), `imul` (multiplication), `idiv` (division)
  - Prennent deux opérandes
  - Affectent le résultat au second opérande
  - `add $2, %rax` : réalise `%rax <- %rax + 2`
- Opérations arithmétiques unaires
  - `inc` (incrémenter), `dec` (décrémenter), `neg` (négation)
- Calcule d'adresse
  - `lea` calcule la valeur d'une adresse indirecte et la range dans le second opérande
- Opérations logiques
  - `and` (et), `or` (ou), `xor` (ou exclusif), `not` (négation logique)
  - Décalage de bit : `sal` (vers la gauche), `shr` (vers la droite)
- Comparaison (sans modification de registre) : `cmp`

Opérations de branchement

- Les instructions arithmétiques et logiques positionnent des drapeaux
  - ZF : le résultat vaut zéro
  - CF : le résultat a provoqué une retenue au delà du bit de poids fort
  - SF : le résultat est négatif en arithmétique signée
  - OF : le résultat a provoqué un débordement en arithmétique signée
- `jz L` : saute à l'étiquette `L` si `ZF` indique nul
  -

`jz`	$= 0$	`jg`	$>$ signé	`ja`	$>$ non signé
`jnz`	$\neq 0$	`jge`	$\leq$ signé	`jae`	$\geq$ non signé
`js`	< 0	`jl`	$<$ signé	`jb`	$<$ non signé
`jns`	≥ 0	`jle`	$\leq$ signé	`jbe`	$\geq$ non signé

- Saut inconditionnel : `jump`

Manipulation de la pile

- `%rsp` : pointe sur le sommet de la pile (croît dans le sens des adresses décroissantes)
- `push` : empile
  - `pushq $42` empile 64 bits correspondant à l'entier 42
- `pop` : dépile
  - `popq %rdi` dépile 64 bits et les écrit dans `%rdi`
- manipulation explicite : `addq $48, %rsp` dépile d'un coup 48 octets

Appel de fonction ♪♪♪

- `call g` : empile l'adresse de l'instruction située après le `call` et transfère le contrôle à l'adresse `g`
- `ret` : dépile une adresse et y transfère le contrôle
- Tout registre modifié par l'appelé sera potentiellement perdu pour l'appelant. Le contenu des registres doivent être sauvegardés dans la pile : tableau d'activation
- Le tableau d'activation contient aussi : des arguments (si plus de 6), des variables locales...

Appel de fonction : convention

- Les six premiers paramètres sont passés dans les registres `%rdi, %rsi, %rdx, %rcx, %r8, %r9`
- Les autres arguments sont passés dans la pile
- La valeur de retour est passée dans `%rax`
- Les registres `%rbx, %rbp, %r12, %r13, %14 et %r15` : *callee-saved* (l'appelé doit les sauvegarder)
- Les autres registres : *caller-saved* (l'appelant doit les sauvegarder)
- La taille du tableau n'étant pas fixe : coutume d'utiliser `%rbp` pour désigner le début du tableau d'activation
- L'ancienne valeur de `%rbp` doit elle-même faire partie du tableau d'activation

Appel de fonction déroulement

- Avant l'appel, l'appelant
  - passe les arguments dans `%rdi, %rsi, %rdx, %rcx, %r8, %r9`
  - et les autres dans la pile
  - sauvegarde les registres *caller-saved*
  - exécute l'instruction `call`
- Au début de l'appel, l'appelé
  - sauvegarde `%rbp` puis le positionne (`pushq %rbp` puis `movq %rsp, %rbp`)
  - alloue son tableau d'activation (diminue la valeur de `%rsp`)
  - sauvegarde les registres *callee-saved*
- A la fin de l'appel, l'appelé
  - place le résultat dans `%rax`
  - restaure les registres *callee-saved* sauvegardés
  - dépile son tableau d'activation
  - restaure `%rbp` (`movq %rbp, %rsp` puis `popq %rbp`)
  - appel l'instruction `ret`
- Après l'appel, l'appelant
  - dépile les éventuels arguments
  - restaure les registres *caller-saved*

RAT → X86-64 : 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 plus1(x, y);
}
```

Allocation de registre

- Les deux paramètres de `plus1` sont alloués dans les registres `%rdi` et `%rsi`
- Comme le résultat est dans `%rax`, on choisit d'allouer la variable locale `c` dans ce registre
- Les variables locales du programme principal peuvent être mises dans `%r10` et `%r11`

Code de la fonction plus1

```
plus1 :
   pushq  %rbp          # sauvegarde %rbp
   movq   %rsp,  %rbp   # positionnement de %rbp
                        # allocation du tableau d'activation (NOP)
                        # sauvegarde les registres callee-saved

   movq   $1,    %rax   # positionne 1 dans le registre %rax
   addq   %rsi,  %rax   # %rax <- %rsi + %rax (c <- b+c)
   addq   %rdi,  %rax   # %rax <- %rdi + %rax (c <- a+(b+c))

                        # restaure les registres callee-saved
                        # dépile le tableau d'activation (NOP)
   movq   %rbp,  %rsp   # restaure %rbp
   popq   %rbp
   ret
```

Code du bloc principal

```
main :
   movq   $3,    %r10  # positionne 3 dans le registre %r10
   movq   $4,    %r11  # positionne 4 dans le registre %r11

   movq   %r10,  %rdi  # positionne l'argument a
   movq   %r11,  %rsi  # positionne l'argument b
   call   plus1        # rien à sauvegarder
                       # rien à restaurer
                       # retour dans %rax
   movq   $Sprint, %rdi
   movq   %rax, %rsi
   xorq   %rax, %rax
   call   printf

.data
   Sprint:
          .string "%d\n"
```

Test

```
> clang exemple.s -o exemple
> ./exemple
8
```

Bilan

- Exemple simple (pas de fonction récursive) ne nécessitant pas de sauvegarde manuelle du contexte d'appel
- Nécessite un travail sur l'allocation de registre.
- Peut être optimisé (sauvegarde du sommet de pile inutile par exemple).

RAT → X86-64 : Exemple de la factorielle

```
int fact (int i, int n) {
   int res = 0;
   if (i=n) {
       res = i;
   } else {
       res = (i * fact((i+1), n));
 }
 return res; }

test {
   int x = fact(1, 5);
   print x; }
```

Allocation de registre

- fact
  - i : `%rdi`, évolue dans les appels récursif, doit être sauvegardé
  - n : `%rsi`, n'évolue pas
  - res : `%rax`, n'évolue pas
- main
  - x : `%r10`

RAT → X86-64 : Exemple de la factorielle

```
   .text                # indique que ce qui suit est un programme
   .globl  main         # rend le symbole main visible, début exécutable

fact :
   pushq  %rbp          # sauvegarde %rbp
   movq   %rsp,  %rbp   # positionnement de %rbp

   movq   $0,    %rax   # positionne 0 dans le registre %rax
   cmpq   %rdi,  %rsi   # compare i (%rdi) et n (%rsi)
   jnz    neq

eq :
   movq  %rdi,  %rax    # res = i (%rax <- %rdi)
   jmp   fin

neq :
   pushq  %rdi          # sauvegarde des registres caller-saved
   incq   %rdi          # i = i+1
   call   fact          # résultat de %rax
   popq   %rdi          # restauration des registres caller-saved
   imulq  %rdi,  %rax   # res <- i * appel rec

fin :
   movq   %rbp,  %rsp   # restaure %rbp
   popq   %rbp
   ret
```

RAT → X86-64 : Exemple de la factorielle

```
main :
   movq   $1,    %r10  # positionne 1 dans le registre %r10
   movq   $5,    %r11  # positionne 5 dans le registre %r11
   movq   %r10,  %rdi  # positionne l'argument a
   movq   %r11,  %rsi  # positionne l'argument b
   call   fact         # retour dans %rax
   movq   $Sprint, %rdi
   movq   %rax, %rsi
   xorq   %rax, %rax
   call   printf

.data
   Sprint:
          .string "%d\n"
```

Test

```
> clang fact.s -o fact
> ./fact
120
```

Bilan

- Présentation d'un appel récursif avec sauvegarde d'un registre *caller-saved*.
- Nécessite un travail sur l'allocation de registre.
- Nécessite une analyse des registres qui doivent être sauvegardés.
- Peut être optimisé (sauvegarde du sommet de pile inutile par exemple).

Quelques mots sur l'allocation de registre

- Associer un *temporaire* à chaque variable
- Réduire le nombre de temporaires grâce à une analyse de durée de vie (ie. le contenu des temporaires ne se mélangent pas)
- Plus de temporaires que de registres disponibles ?
  $\rightarrow$ temporaires placés dans la pile
- But de l'allocation de registre :
  - minimiser le nombre de temporaires alloués en pile
  - minimiser les transferts entre registres
- Méthode :
  - Réalisation d'un graphe d'interférence entre les temporaires (sommet : temporaires, arc : sommets pas dans même registre)
  - Allouer des registres aux temporaires revient à colorier le graphe d'interférence
  - Problème NP-complet...
  - ... mais algorithme simple (et linéaire) permettant de colorier un graphe avec K couleurs (K : nombre de registres)

Septième partie VII

Conclusion

Conclusion

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.
  $\Rightarrow$ Souci : 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 informations sur les symboles non résolus.
  $\Rightarrow$ Souci : compilation et exécution doivent être faites dans des environnements compatibles.

Conclusion

Écriture d'un compilateur pour un langage simple

1. Donner la grammaire du langage ;
2. Écrire 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).

Écriture d'un compilateur pour un langage complexe

Toutes les phases de la compilation seront nécessaires (linéarisation, sélection d'instructions, allocation de registres, optimisation, édition de liens...).

Conclusion

Au-delà de la compilation

La traduction des langages concerne toute tranformation d'un texte source respectant un format d'entrée dans un texte cible respectant un format de sortie.

- Analyses lexicale et syntaxique : comme précédemment
- Analyse sémantique : dépend du traitement à réaliser et du format cible