Traduction des langages

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



Plan

I. Introduction

- 1 Introduction à la compilation
- 2 Les points abordés

II. Analyse lexicale : Rappels et compléments

- 1 Rappels et compléments : Automates et expressions régulières
 - **Automates**
 - Expressions Régulières
- 2 Compléments : Analyse lexicale
- 3 Bilan



Plan

III. Analyse syntaxique

- Grammaire formelle
 - Dérivation
 - Grammaire algébrique
- 2 Arbre de dérivation
- 3 Analyseur syntaxique
- 4 Bilan

IV. Grammaires attribuées

V. Arbre syntaxique abstrait (AST)



Plan

VI. Analyse sémantique : Application à la compilation

- 1 Langage RAT
- 2 Table des symboles
- 3 Typage
- 4 Placement mémoire
- 5 Génération de code

VII. Conclusion



Première partie I

Introduction

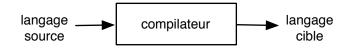


Introduction

Objectif du cours

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

Qu'est ce qu'un compilateur?





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

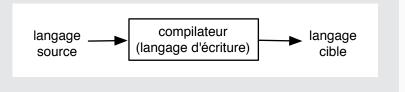
Un programme, qui traduit un programme écrit dans un langage L dans un programme écrit dans un langage L' différent de L (en général L est un langage évolué et L' est un langage de plus bas niveau).



Compilateur

Remarque

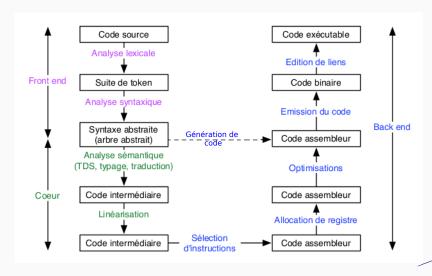
Trois langages : langage source, langage cible et langage d'écriture du compilateur.



Le langage cible peut être :

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





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



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 (cf cours de modélisation de 1A)

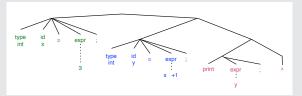


Front-end - Exemple

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



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





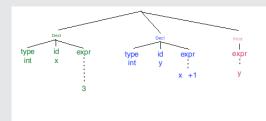
Cœur : Analyse sémantique

- 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
 - · parcours de l'AST
- ⇒ 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



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);
```



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 machines 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'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 - 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
```



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.



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.

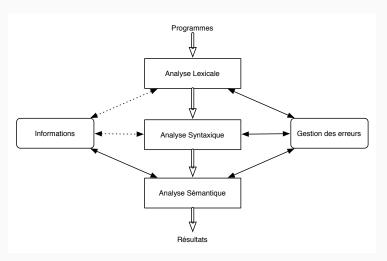


Les points abordés

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







Deuxième partie II

Analyse lexicale : Rappels et compléments

111



régulières

Rappels et compléments :

Automates et expressions

Automate fini (déterministe) à état

Automate fini (déterministe) à état

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

- Q: ensemble fini d'états
- X : alphabet
- $q_l \in Q$: 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 $q' = \delta(q, a)$.

⊢* fermeture réflexive transitive de ⊢

Langage accepté

$$L(A) = \{ m \in X^* \mid \hat{\delta}(q_l, m) \in F \}$$

= $\{ m \in X^* \mid \exists q_F \in F : (q_l, m) \vdash^* (q_F, \Lambda) \}$



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 :

$$L_1 \cup L_2 \in \mathsf{RAT}$$
 $L_1 \bullet L_2 \in \mathsf{RAT}$ $L_1^* \in \mathsf{RAT}$ $L_1 \cap L_2 \in \mathsf{RAT}$ $\overline{L_1} \in \mathsf{RAT}$ $L_1 \setminus L_2 \in \mathsf{RAT}$



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

$$L(u+v) = L(u) \cup L(v)$$

$$L(u \bullet v) = L(u)L(v)$$

$$L(u^*) = L(u)^*$$



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

- 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

 associer un code aux terminaux.





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 Ocamilex

```
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 }
l 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

 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

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 ∈ V est l'axiome de départ;
- P est un sous-ensemble fini de (V ∪ X)⁺ × (V ∪ X)*. Un élément de P est une production et est noté u → v.



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 <u>se dérive</u> 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 $(u \rightarrow v) \in P$.

$$\overset{*}{\Rightarrow}$$

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

Langage engendré par G

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



Exemple de dérivation

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



Grammaire algébrique

Grammaire algébrique

Une grammaire formelle G est <u>algébrique</u> (ou <u>context-free</u> ou <u>non-contextuelle</u>) 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* → { *ListeInst* }
 - 2. ListeInst → 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



Arbre de dérivation

Arbre de dérivation

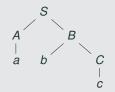
Création de l'arbre de dérivation

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

$$A_1$$
 A_2 \cdots A_n

Soit
$$G: \left\{ egin{array}{lll} 1. & S & \rightarrow AB \\ 2. & A & \rightarrow a \\ 3. & B & \rightarrow bC \\ 4. & C & \rightarrow c \end{array} \right.$$

 $abc \in L(G)$: $\underline{S} \stackrel{1}{\Rightarrow} \underline{AB} \stackrel{2}{\Rightarrow} \underline{aB} \stackrel{3}{\Rightarrow} \underline{abC} \stackrel{4}{\Rightarrow} \underline{abc}$ $\underline{S} \stackrel{1}{\Rightarrow} \underline{AB} \stackrel{3}{\Rightarrow} \underline{AbC} \stackrel{4}{\Rightarrow} \underline{abC} \stackrel{4}{\Rightarrow} \underline{abc}$ $S \stackrel{1}{\Rightarrow} \underline{AB} \stackrel{3}{\Rightarrow} \underline{AbC} \stackrel{4}{\Rightarrow} \underline{Abc} \stackrel{2}{\Rightarrow} \underline{abc}$





Ambiguïté

Deux arbres pour un même mot

Soit
$$G: \left\{ \begin{array}{ll} 1. & E \rightarrow E + E \\ 2. & E \rightarrow a \end{array} \right.$$

$$G': \left\{ \begin{array}{ll} 1. & E & \rightarrow T \ a \\ 2. & T & \rightarrow a + T \\ 3. & T & \rightarrow \Lambda \end{array} \right. \qquad G'': \left\{ \begin{array}{ll} 1. & E & \rightarrow E + T \\ 2. & E & \rightarrow a \\ 3. & T & \rightarrow a \end{array} \right.$$

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



Ambiguïté

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 *G* suivante :

- 1. $A \rightarrow aAb$
- 2. $A \rightarrow AA$
- 3. $A \rightarrow bAa$
- 4. $A \rightarrow \Lambda$

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?

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



^{1.} Quand vous avez le choix entre plusieurs terminaux à dériver, vous dérivez celui qui est le plus à droite.

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. $\{a^nb^p \mid n \ge p \ge 0\}$
- 3. le langage des expressions bien parenthésées, par exemple $(([\bullet][(\bullet)(\bullet)])(\bullet))$



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.



Analyseur syntaxique

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
 ajout de contrainte sur la grammaire pour que ça soit simple et
 efficace.



- 1. $S \rightarrow AB$
- 2. *A* → *a A*
- 3. $B \rightarrow b B$
- 4. $A \rightarrow a$
- 5. $B \rightarrow b$

Ascendant

Lecture : aaabb

aaabb

aa ab b



- 1. $S \rightarrow AB$
- 2. *A* → *a A*
- 3. $B \rightarrow b B$
- 4. $A \rightarrow a$
- 5. $B \rightarrow b$

Ascendant

Lecture : aaabb

aaabb

aa ab b

- 1. $S \rightarrow AB$
- 2. *A* → *a A*
- 3. $B \rightarrow b B$
- 4. $A \rightarrow a$
- 5. $B \rightarrow b$

Ascendant

Lecture : aaabb

aaabb

aa ab b



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

Ascendant

Lecture : *aaabb aaabb* ^⁴ *aaAbb*



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

Ascendant

Lecture : aaabb

 $aaabb \stackrel{\scriptscriptstyle 4}{\Leftarrow} aaAbb \stackrel{\scriptscriptstyle 2}{\Leftarrow} aAbb$



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

Ascendant

Lecture : aaabb aaabb ← aaAbb ← aAbb ← Abb





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

Ascendant



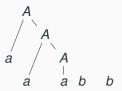


- 1. $S \rightarrow AB$
- 2. $A \rightarrow a A$
- 3. $B \rightarrow b B$
- 4. $A \rightarrow a$
- 5. $B \rightarrow b$

Ascendant

Lecture : aaabb

aaabb ^⁴ aaAbb ^² aAbb ^² Abb





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

Ascendant

Lecture : aaabb

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





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

Ascendant

Lecture : aaabb

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



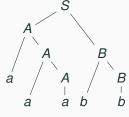


- 1. $S \rightarrow AB$
- 2. $A \rightarrow a A$
- 3. $B \rightarrow b B$
- 4. $A \rightarrow a$
- 5. $B \rightarrow b$

Ascendant

Lecture : aaabb

 $aaabb \stackrel{4}{\leftarrow} aaAbb \stackrel{?}{\leftarrow} aAbb \stackrel{?}{\leftarrow} Abb \stackrel{5}{\leftarrow} AbB \stackrel{3}{\leftarrow} AB \stackrel{1}{\leftarrow} S$



Dérivation la plus à droite



- 1. $S \rightarrow AB$
- 2. *A* → *a A*
- 3. $B \rightarrow b B$
- 4. $A \rightarrow a$
- 5. $B \rightarrow b$

- Descendant
 - <u>S</u>

S

- 1. $S \rightarrow AB$
- 2. *A* → *a A*
- 3. $B \rightarrow b B$
- 4. $A \rightarrow a$
- 5. $B \rightarrow b$

Descendant

$$\underline{S} \stackrel{1}{\Rightarrow} \underline{A}B$$

$$S$$

$$A$$

$$F$$



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

Descendant

$$\underline{S} \stackrel{1}{\Rightarrow} \underline{A}B \stackrel{2}{\Rightarrow} a\underline{A}B$$



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

Descendant

$$\underline{S} \stackrel{1}{\Rightarrow} \underline{A}B \stackrel{2}{\Rightarrow} a\underline{A}B \stackrel{2}{\Rightarrow} aa\underline{A}B$$



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

Descendant

 $\underline{S} \stackrel{1}{\Rightarrow} \underline{A}B \stackrel{2}{\Rightarrow} a\underline{A}B \stackrel{2}{\Rightarrow} aa\underline{A}B \stackrel{4}{\Rightarrow} aaa\underline{B}$

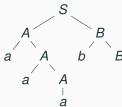




- 1. $S \rightarrow AB$
- 2. $A \rightarrow a A$
- 3. $B \rightarrow b B$
- 4. $A \rightarrow a$
- 5. $B \rightarrow b$

Descendant

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





- 1. $S \rightarrow AB$
- 2. $A \rightarrow a A$
- 3. $B \rightarrow b B$
- 4. *A* → *a*
- 5. $B \rightarrow b$

Descendant

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



Dérivation la plus à gauche

Analyse syntaxique

Il est possible de 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
 - ...



Analyse syntaxique

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



Analyse syntaxique

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)

```
%token INT
%token NAME
%token XXXX
%token EOF
(* Axiome *)
%start main
응응
main:
| ligne EOF {()}
| ligne main {()}
ligne :
I INT NAME INT INT {()}
I INT NAME XXXX INT {()}
```



Bilan

Bilan

- La phase d'analyse lexicale permet de découper le programme source en tokens qui correspondent aux terminaux de la grammaire.
- 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, X₁,..., 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 \{VS.liste = []\}

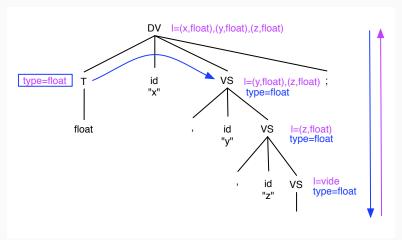
4. T \rightarrow int \{T.stype = int\}

5. T \rightarrow float \{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 : $A \rightarrow X_1 \dots X_n \quad \{X_i.h \leftarrow \dots\}$.

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 : $A \rightarrow X_1 \dots X_n \quad \{A.s \leftarrow \dots\}.$

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.

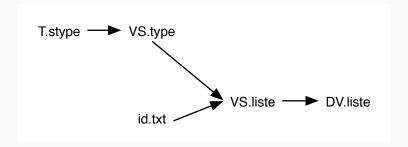
Grammaire attribuée

Grammaire + attributs sémantiques + actions sémantiques



Evaluation 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



Menhir

Attributs et actions dans Menhir

- Menhir : analyseur ascendant ⇒ 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 attribut sur les terminaux, positionnés par l'analyseur lexical
 - Syntaxe: %token <type de l'att> <terminal>



```
%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
응 응
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::m} : action sémantique permettant de calculer la valeur de
  l'attribut synthétisé du "main" parent *)
liane :
| sexe = INT prenom = NAME annee = INT nb = INT { (sexe, prenom, annee, nb) }
| sexe = INT prenom = NAME XXXX nb = INT { (sexe, prenom, -1, nb) }
```

,

71/170

Bilan

Bilan

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

- 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.
- ⇒ 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

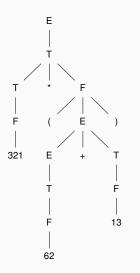
Struture des arbres

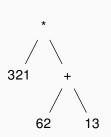
- L'arbre de dérivation est unique pour une entrée donnée et une analyse, alors que l'AST est indépendant de l'analyse.
- Les nœuds de l'arbre de dérivation sont imposés par la grammaire, alors que l'on peut choisir le type des nœuds de l'arbre abstrait.



Exemple : Arbre de dérivation vs Arbre abstrait

Source : 321 * (62 + 13)







Construction de l'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{split} E \rightarrow E_1 \ + \ T \quad & \{E.ast = \textit{Node}("+", E_1.ast, T.ast)\} \\ E \rightarrow T \quad & \{E.ast = T.ast\} \\ T \rightarrow T_1 \ * \ F \quad & \{T.ast = \textit{Node}("*", T_1.ast, F.ast\} \\ T \rightarrow F \quad & \{T.ast = F.ast\} \\ F \rightarrow \textit{entier} \quad & \{F.ast = \textit{Node}(\textit{entier.val})\} \\ F \rightarrow & (E) \quad & \{F.ast = E.ast\} \end{split}
```



Bilan

Bilan

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



Langage RAT

Langage RAT

La grammaire

- 1. PROG' → PROG\$
- 2 PROG → FUN PROG
- 3. $FUN \rightarrow TYPE id (DP) \{IS \ return \ E; \}$ 20. $CP \rightarrow \Lambda$
- 4. PROG → 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. $I \rightarrow \text{ while } E \text{ BLOC}$
- 14. $DP \rightarrow \Lambda$
- 15. $DP \rightarrow TYPE id DP$
- 16. $TYPE \rightarrow bool$
- 17 $TYPF \rightarrow int$

- 18. $TYPE \rightarrow rat$
- 19. $E \rightarrow call id (CP)$
- 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 $F \rightarrow false$
- 28. $E \rightarrow entier$
- 29. $E \rightarrow (E + E)$
- 30. $E \rightarrow (E * E)$
- 31. $E \rightarrow (E = E)$
- 32. $E \rightarrow (E < E)$
- 33. $E \rightarrow (E)$



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(call less(a b)) {print a;} else {print b;}
```

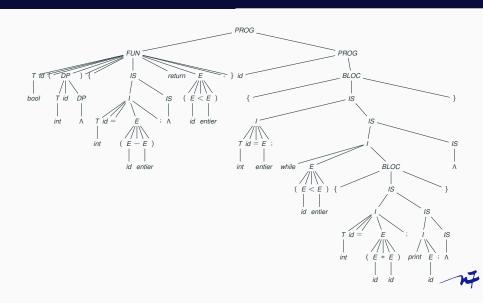
Langage RAT

Un exemple simple

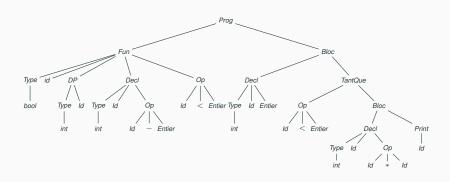
```
bool less ( int a ) {
int b = (a-3);
return (b<0);
prog{
 int i = 0;
 while(i<5){
   int j = (i*i);
  print j;
```



Arbre de dérivation



Un arbre abstrait possible





Langage RAT : arbre abstrait

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 ⇒ permet l'utilisation du pattern matching

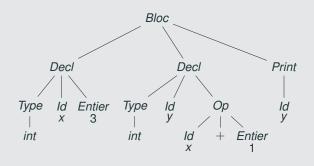


Table des symboles

code

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

AST





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

Opérations sur la TDS

- Créer la TDS
- · Insérer un symbole (nom, information)
- Chercher un symbole (nom → information)

Exemples d'implantation

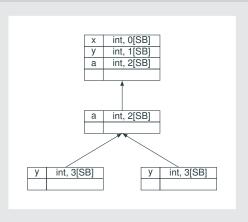
- Tableau
- · Liste chaînée
- · Arbre binaire de recherche
- Table de hachage
- . . .





Portée des variables

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



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

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 Nationale 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
         \rightarrow 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 entier : int \qquad \sigma \vdash true : bool \qquad \sigma \vdash false : bool
```



Accès à l'environnement

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

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

777

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

Définition locale

$$\frac{\sigma \vdash e_1 : \tau_1 \qquad \sigma :: \{X : \tau_1\} \vdash e_2 : \tau_2}{\sigma \vdash \text{let } X = e_1 \text{ in } e_2 : \tau_2}$$



Typage - Jugement de typage - Mini-ML

Définition de fonction

111

$$\frac{\sigma :: \{X : \tau_1\} \vdash e : \tau_2}{\sigma \vdash \text{fun } X \rightarrow e : \tau_1 \rightarrow \tau_2}$$

Appel de fonction

444

$$\frac{\sigma \vdash \mathbf{e}_1 : \tau_1 \to \tau_2 \qquad \sigma \vdash \mathbf{e}_2 : \tau_1}{\sigma \vdash (\mathbf{e}_1 \ (\mathbf{e}_2)) : \tau_2}$$

Définition récursive

$$\frac{\sigma :: \{X : \tau_1\} \vdash e_1 : \tau_1 \qquad \sigma :: \{X : \tau_1\} \vdash e_2 : \tau_2}{\sigma \vdash \text{let rec } X = e_1 \text{ in } e_2 : \tau_2}$$



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 $\frac{ }{\{\} \vdash \text{let } x = 1 \text{ in } x + 1 : \tau }$



Exemple - Définition locale





Exemple - Définition locale

$$\frac{\{x: int\} \vdash x: \tau_1 \quad \{x: int\} \vdash 1: \tau_2 \quad \tau_1 \times \tau_2 = dom + \quad \tau = codom + \\ \{x: int\} \vdash x + 1: \tau \\ \{\} \vdash \text{let } x = 1 \text{ in } x + 1: \tau$$



Exemple - Définition locale

$$\frac{\{x: int\} \vdash x: int \ \ \, \{x: int\} \vdash 1: \tau_2 \ \ \, \overline{int \times \tau_2 = dom + \ \, \tau = codom + \ \, }}{\{x: int\} \vdash x: 1: \tau}$$

$$\{\} \vdash \text{let } x = 1 \text{ in } x + 1: \tau$$



Exemple - Définition locale

$$\frac{\{x: int\} \vdash x: int \quad \overline{\{x: int\} \vdash 1: int \quad int \times int = dom + \quad \tau = codom + \quad \{x: int\} \vdash x + 1: \tau}{\{\} \vdash \text{let } x = 1 \text{ in } x + 1: \tau}$$



Exemple - Définition locale

$$\frac{\{x: int\} \vdash x: int \quad \overline{\{x: int\}} \vdash 1: int \quad int \times int = dom + \quad \overline{int} = codom + \\ \{\} \vdash 1: int \quad \{x: int\} \vdash x + 1: int \\ \{\} \vdash \text{let } x = 1 \text{ in } x + 1: int$$



Exemple - Définition de fonction et conditionnelle 333



Exemple - Définition de fonction et conditionnelle

444

 $\{\,\} \ \vdash \ \text{fun $a \to $fun $b \to $fun $c \to $if a then b else $c : $\tau_1 \to \tau_2$}$



Exemple - Définition de fonction et conditionnelle

444

```
\{a: \tau_1\} \vdash \text{fun } b \rightarrow \text{fun } c \rightarrow \text{if } a \text{ then } b \text{ else } c: \tau_2
```

 $\{\,\} \ \vdash \ \operatorname{fun} a \mathbin{{\scriptscriptstyle{-}}{\scriptscriptstyle{>}}} \operatorname{fun} b \mathbin{{\scriptscriptstyle{-}}{\scriptscriptstyle{>}}} \operatorname{fun} c \mathbin{{\scriptscriptstyle{-}}{\scriptscriptstyle{>}}} \operatorname{if} a \operatorname{then} b \operatorname{else} c \ : \ \tau_1 \ \to \ \tau_2$



Exemple - Définition de fonction et conditionnelle

444

$$\{a: \tau_1\} \vdash \operatorname{fun} b \rightarrow \operatorname{fun} c \rightarrow \operatorname{if} a \operatorname{then} b \operatorname{else} c: \tau_3 \rightarrow \tau_4$$

 $\{\,\} \ \vdash \ \text{fun $a \to $fun $b \to $fun $c \to $if a then b else $c : $\tau_1 \to \tau_3 \to \tau_4$}$



Exemple - Définition de fonction et conditionnelle



Exemple - Définition de fonction et conditionnelle



Exemple - Définition de fonction et conditionnelle



Exemple - Définition de fonction et conditionnelle



Exemple - Définition de fonction et conditionnelle



Exemple - Définition de fonction et conditionnelle



Exemple - Définition de fonction et conditionnelle



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 → 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 (call foo(a));
}</pre>
```

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.
- . . .

Sémantique naturelle du typage

444

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

444

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

Base

- $(\mathbf{X}, \tau) :: \sigma \vdash \mathbf{X} : \tau$
- $\sigma \vdash \mathbf{x} : \tau$
- $\overline{(y,\tau')}:: \sigma \vdash x : \tau$

- $\sigma \vdash true : bool$
- σ ⊢ false : bool
- $\sigma \vdash$ entier : int



Les cas d'erreur ne seront pas donnés.

Expression (1)

•
$$\frac{\sigma \vdash E_1 : int \qquad \sigma \vdash E_2 : int}{\sigma \vdash [E_1 / E_2] : rat}$$

$$\sigma \vdash E : rat$$

$$\overline{\sigma \vdash \mathit{num} \; E : \mathit{int}}$$

•
$$\frac{\sigma \vdash E : rat}{\sigma \vdash denom \ E : int}$$

•
$$\frac{\sigma \vdash E : \tau}{\sigma \vdash (E) : \tau}$$



Expression (2)

•
$$\frac{\sigma \vdash E_1 : int \quad \sigma \vdash E_2 : int}{\sigma \vdash (E_1 + E_2) : int}$$
•
$$\frac{\sigma \vdash E_1 : rat \quad \sigma \vdash E_2 : rat}{\sigma \vdash (E_1 + E_2) : rat}$$
•
$$\frac{\sigma \vdash E_1 : int \quad \sigma \vdash E_2 : int}{\sigma \vdash (E_1 * E_2) : int}$$
•
$$\frac{\sigma \vdash E_1 : rat \quad \sigma \vdash E_2 : rat}{\sigma \vdash (E_1 * E_2) : rat}$$

•
$$\frac{\sigma \vdash E_1 : int \quad \sigma \vdash E_2 : int}{\sigma \vdash (E_1 = E_2) : bool}$$

•
$$\frac{\sigma \vdash E_1 : bool \quad \sigma \vdash E_2 : bool}{\sigma \vdash (E_1 = E_2) : bool}$$

$$\frac{\sigma \vdash E_1 : int \quad \sigma \vdash E_2 : int}{\sigma \vdash (E_1 < E_2) : bool}$$

On se limitera à ces signatures >>>.



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, []}
```

$$\frac{\sigma \vdash E : bool}{\sigma \vdash while \ E \ BLOC : void}$$



Déclaration / affectation

777

•
$$\frac{\sigma \vdash \mathit{TYPE} : \tau \qquad \sigma \vdash \mathit{E} : \tau}{\sigma \vdash \mathit{TYPE} \; \mathit{id} = \mathit{E} : \mathit{void}, [\mathit{id}, \tau]}$$

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

Instructions

•
$$\sigma \vdash const \ id = entier : void, [id, int]$$

$$\sigma \vdash E : \tau$$

$$\sigma \vdash print \ E : void, []$$



Déclaration de fonction

111

- $\overline{\sigma \vdash TYPE \ id \ (DP) \ \{IS \ return \ E;\}} : void, [id, \tau_2 \rightarrow \tau_1]$
 - A : $\sigma \vdash TYPE : \tau_1$
 - B: $\sigma \vdash DP : \tau_2, \sigma_p$ (DP = TYPE₁ id₁ . . . TYPE_n id_n)
 - C: $(id, \tau_2 \rightarrow \tau_1) \circ \sigma_p \circ \sigma \vdash IS$: $void, \sigma_I$
 - D: $(id, \tau_2 \rightarrow \tau_1) @\sigma_I @\sigma_p @\sigma \vdash E : \tau_1$

•

$$\sigma \vdash TYPE_1 : \tau_1 \quad \dots \quad TYPE_n : \tau_n$$

$$\overline{\sigma \vdash TYPE_1 \ id_1 \ \dots \ TYPE_n \ id_n : \tau_1 \times \dots \times \tau_n, [(id_1, \tau_1); \dots; (id_n, \tau_n)]}$$



Appel de fonction

•
$$\frac{\sigma \vdash \textit{id} : \tau_1 \rightarrow \tau_2 \qquad E_1 \ \dots \ E_n : \tau_1}{\sigma \vdash \textit{call id} \ (E_1 \ \dots \ E_n) : \tau_2}$$

•
$$\frac{\sigma \vdash E_1 : \tau_1 \dots E_n : \tau_n}{\sigma \vdash E_1 \dots E_n : \tau_1 \times \dots \times \tau_n}$$



Suite d'instructions

777

•
$$\frac{\sigma \vdash \mathit{IS} : \mathit{void}, \sigma'}{\sigma \vdash \{\mathit{IS}\} : \mathit{void}}$$

$$\bullet \ \frac{\sigma \vdash \textit{I} : \textit{void}, \sigma' \qquad \sigma' @ \sigma \vdash \textit{IS} : \textit{void}, \sigma''}{\sigma \vdash \textit{I} \: \textit{IS} : \textit{void}, \sigma'' @ \sigma'}$$

• $\sigma \vdash$: void

Le programme

$$\bullet \ \frac{\sigma \vdash \mathit{FUN} : \mathit{void}, \sigma' \qquad \sigma' @ \sigma \vdash \mathit{PROG} : \mathit{void}, \sigma''}{\sigma \vdash \mathit{FUN} \ \mathit{PROG} : \mathit{void}, \sigma'' @ \sigma'}$$

•
$$\frac{\sigma \vdash BLOC : void}{\sigma \vdash id \ BLOC : void}$$

Remarque : l'identifiant du bloc 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 - Contrôle de type - RAT

```
Typage de l'exemple
                                                                                                                                                                                                                                                                В
                  [] \vdash bool\ foo(int\ x)\{...\}: void, [foo.int \rightarrow bool] [foo.int \rightarrow bool] \vdash proa\{...\}: void, []
   [] \vdash bool\ foo(int\ x) \{ int\ y = 1;\ return\ (x < y); \} prog \{ int\ a = 4;\ print(call\ foo(a)); \} : void, []
 A :
                                                                                                                                                                                                                                                                                                                                      [x, int] \vdash x : int
                                                                                                                                                                                                                                                                                                                                                                                                                   [y, int] \vdash y : int
                                                                         \prod \vdash int : int
                                                                                                                                                                                [..] \vdash int : int [..] \vdash 1 : int
                                                                                                                                                                                                                                                                                                                           [...; (x,int)] \vdash x : int [...; (y,int); ...] \vdash y : int
[] \vdash bool:bool:] \vdash int x:int,[x,int] [(foo,int \rightarrow bool);(x,int)] \vdash int y = 1:void,[y,int] [(foo,...);(y,int);(x,int)] \vdash (x < y):bool:bool:[] \vdash int x:int,[x,int] [(foo,int \rightarrow bool);(x,int)] \vdash int y = 1:void,[y,int] [(foo,...);(y,int);(x,int)] \vdash (x < y):bool:bool:[] \vdash int x:int,[x,int] [(foo,int \rightarrow bool);(x,int)] \vdash int y = 1:void,[y,int] [(foo,int)] [
                                                                                                         [] \vdash bool foo(int x) \{ int y = 1; return (x < y); \} : void, [foo,int <math>\rightarrow bool]
  B :
                                                                                                                                                                                                                    \lceil foo, int \rightarrow bool \rceil \vdash foo: int \rightarrow bool \rceil
                                                                                                                                                                                                         [...; (foo,int \rightarrow bool)] \vdash foo:int \rightarrow bool [(a,int); ...] \vdash a:int
                                                                                                                                                                                                                                  [(a,int); (foo,int \rightarrow bool)] \vdash call foo(a);:bool
            \lceil foo, int \rightarrow boof \rceil \vdash int : int \lceil foo, int \rightarrow boof \rceil \vdash 4 : int \rceil
                                                                                                                                                                                                                              [(a,int); (foo,int \rightarrow bool)] \vdash (call foo(a));:bool
                             [foo, int \rightarrow bool] \vdash int a = 4; void, [a, int]
                                                                                                                                                                                                               [(a,int); (foo,int \rightarrow bool)] \vdash print(call foo(a));: void, []
                                                                                                                [foo, int \rightarrow boof] \vdash int a = 4: print(call foo(a)):: void, []
                                                                                                         [foo, int \rightarrow bool] \vdash \{ int a = 4; print(call foo(a)); \} : void, []
                                                                                              [foo, int \rightarrow bool] \vdash prog\{int a=4; print(call 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?

```
• NN
Déclaration :

\frac{\sigma \vdash TYPE : \tau_1}{\sigma \vdash TYPE id = E : void, \{id : \tau_1\}}
```

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



Pour aller plus loin...

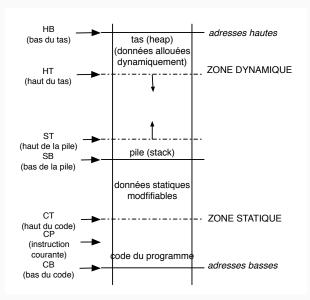
Réfléchissez aux réponses avant d'écouter les audio qui proposent des pistes de solutions.

- · Comment gérer l'héritage de type?
 - NNDéclaration : $\frac{\sigma \vdash TYPE : \tau_1}{\sigma \vdash TYPE \ id = E : void, \{id : \tau_1\}}$
- Comment construire et nommer un nouveau type? (par exemple des couples...)
 - . 111



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.





Variables locales

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

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



Allocation statique

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

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

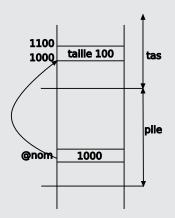


Allocation dynamique

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).



Libération de la mémoire

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

		← CT
Code :		
	5	← CP
	4	
	3	
	2	
	1	
	0	← CB

	5	
	4	← ST
ile :	3	
	2	
	1	
	0	← SB

	999	← HB
	998	
	997	
s:	996	
	995	
	994	\leftarrow HT

Ta



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?



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



```
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;}
```

	← ST	
1		
1	← SB	@a=0[SB]
	1 1	1



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



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



```
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		
2	1		@b=2[SB]
1	1		
0	1	← SB	@a=0[SB]

- num a < 2 → else
- Calculs effectués dans la pile
- Pile nettoyée après calcul



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



```
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



```
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			
7 6 5 4 3 2			
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]



```
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		
2	1		@b=2[SB]
1	1		
0	1	← SB	@a=0[SB]



```
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]



```
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?



```
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;}
```

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

Adresse de a1? → 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));
}
fonction{
  int x = 3;
  int y = 4;
  print call plus1 ((x+1) y);
  int i = 1;
  int j = 2;
  print call plus1 (i j);
}
```

⇒ 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 (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.



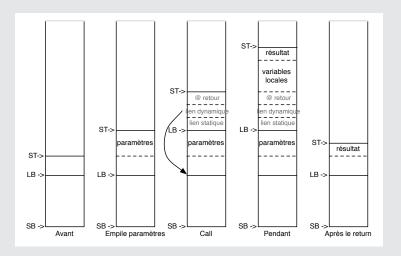
Enregistrement d'activation : l'adresse de retour

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 permet 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 :
 - → 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 :
 - → déplacement positif par rapport à LB.



Retour sur l'exemple

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



Pour aller plus loin...

- Pas nécessairement besoin de pile pour faire l'appel de fonction, si pas de récursivité.
- La pile des variables locales ≠ pile des enregistrements d'activation : mais on traite les deux dans une seule.



Génération de code

Machine TAM 1757

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.

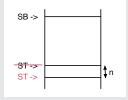
Instructions

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

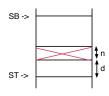
La machine TAM : Instructions agissant sur la zone de code (et la pile)

- 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.

• PUSH n : Agrandit la pile de n mots

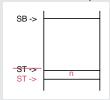


• POP (d) n : Garde les d premiers et supprime les n suivants

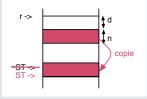




• LOADL n : Empile la valeur n

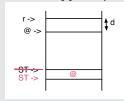


 LOAD (n) d[r]: Copie le bloc de taille n qui est à l'adresse d[r] en sommet de pile

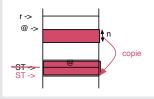




• LOADA d[r] : Empile l'adresse d[r] en sommet de pile

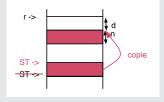


 LOADI (n): Prend l'adresse en sommet de pile, et copie le bloc de taille n de cette adresse en sommet de pile

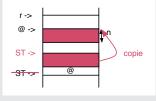




 STORE(n) d[r]: Déplace le bloc de taille n en sommet de la pile à l'adresse d[r]



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

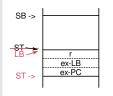




TAM: gestion des fonctions

 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 – on n'en aura a priori pas besoin), l'adresse de l'instruction à exécuter une fois l'appel fini.

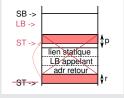
Le registre LB contient la base de l'enregistrement d'activation (le sommet de pile au moment de l'appel).





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 pour les résultats.





$\text{RAT} \to \text{TAM}$: 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);
}
```



$\text{RAT} \to \text{TAM}$: 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
LOADI, 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
```

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

```
plus1 ; fonction plus1
PUSH 1
                ; place pour c
LOADI. 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

Démonstration de Itam.

111



Génération de code

Cas particulier de TAM

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

Pour un assembleur plus évolué?

- x86-64⁴
- On ne traitera pas 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 → registres processeur



Les regi	stres 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 ₁

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)
                           # termine le programme
  ret
   data
                           # signifie que ce qui suit est des données
                           # adresse à laquelle la chaîne est stockée
message:
   .string "hello, world"
```

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

Exécution

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



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 op1 op2: copie de op1 dans op2
 - 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

$$jz = 0$$
 $jg > signé$ $ja > non signé$ $jnz \neq 0$ $jge \leq signé$ $jae \geq non signé$ $js < 0$ $j1$ $< signé$ jb $< non signé$ $jns \geq 0$ $j1e \leq signé$ $jbe \geq non signé$

• 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

444

- 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 sur 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



$RAT \to X86\text{-}64$: 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 call (plus1 x y);
}
```

Allocation de registre

- Les deux paramètres de plus1 seront 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 :
  pushg %rbp # sauvegarde %rbp
  mova
        %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))</pre>
                      # restaure les registres callee-saved
                      # dépile le tableau d'activation (NOP)
         %rbp,
               %rsp # restaure %rbp
  movq
  popq
         %rbp
  ret
```



Code du bloc principal

```
main :
  mova
        $3, %r10 # positionne 3 dans le registre %r10
        $4, %rll # positionne 4 dans le registre %rll
  movq
  movg %r10, %rdi # positionne l'argument a
  movq %rll, %rsi # positionne l'argument b
  call plus1
                     # rien à sauvegarder
                     # rien à restaurer
                     # retour dans %rax
        $Sprint, %rdi
  movq
  movq %rax, %rsi
        %rax, %rax
  xora
  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).



$RAT \rightarrow X86-64$: Exemple de la factorielle

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

test {
  int x = call 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



$RAT \rightarrow X86-64$: Exemple de la factorielle

```
# indique que ce qui suit est un programme
  .text
  .globl main
                    # rend le symbole main visible, début exécutable
fact :
  pushg %rbp # sauvegarde %rbp
        %rsp, %rbp # positionnement de %rbp
  mova
  movq $0, %rax # positionne 0 dans le registre %rax
  cmpq %rdi, %rsi # compare i (%rdi) et n (%rsi)
  jnz
        neq
eq:
  movg %rdi, %rax # res = i (%rax <- %rdi)
  jmp fin
nea:
  pushq %rdi
                    # sauvegarde des registres caller-saved
  incq %rdi
                    \# i = i+1
  call fact # résultat de %rax
  popq %rdi # restauration des registres caller-saved
  imulg %rdi, %rax # res <- i * appel rec
fin:
        %rbp, %rsp # restaure %rbp
  mova
        %rbp
  popq
  ret
```

$RAT \rightarrow X86-64$: Exemple de la factorielle

```
main :
  movq $1, %r10 # positionne 1 dans le registre %r10
  movq $5, %rll # positionne 5 dans le registre %rll
  movg %r10, %rdi # positionne l'argument a
  movq %rll, %rsi # positionne l'argument b
  call fact
               # retour dans %rax
  movg $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?
 - ightarrow 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) ⇒ 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.
 - ⇒ 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.
 - ⇒ 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

S'étend à 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

