

Cours 1 : Qu'est-ce qu'un compilateur?

Yann Régis-Gianas
`yrg@pps.univ-paris-diderot.fr`

PPS - Université Denis Diderot – Paris 7

Qu'est-ce que la compilation ?

- | Vous avez tous déjà tapé :

```
% javac Fact.java
```

- | Que se passe-t-il ensuite ?

Qu'est-ce qu'un compilateur ?

- Il se peut que votre programme soit rejeté :

```
Fact.java:3: incompatible types
found   : boolean
required: int
    if (n == 0) return (true);
                        ^
1 error
```

- Il se peut aussi que votre programme soit accepté.
Dans ce cas, vous pouvez exécuter le programme :

```
% java Fact
2004310016
```

Comment fonctionne un compilateur ?

- | C'est le sujet de ce cours.
- | Aujourd'hui, nous allons tenter de répondre à ces trois questions :
 1. Qu'attend-t-on du compilateur ?
 2. Quels sont les techniques d'implémentation utilisées par la compilation ?
 3. Pourquoi étudier la compilation ?

Qu'attend-t-on du compilateur ?

Qu'attend-t-on du compilateur ?

- | Pour donner une spécification à un compilateur, il faut d'abord bien comprendre ce qu'est la *programmation*.
- | Informellement, la programmation est l'art de résoudre des problèmes efficacement, par le calcul.

Résoudre des problèmes ?

- | Il existe de nombreux types de problèmes : certains sont simples à résoudre et d'autres sont plus difficiles, certains ont une solution connue, d'autres pas, certains ont un énoncé très court tandis que d'autres nécessitent une description gigantesque.
- | Dans tous les cas, cependant, un problème, bien posé, peut se spécifier à l'aide d'une formule logique de la forme :

$$\forall I, P(I) \Rightarrow \exists O, Q(I, O)$$

- | où :
 - ▶ P est la précondition du problème : le domaine de ses entrées valides.
 - ▶ Q est sa postcondition : la relation attendue entre les entrées et les sorties.

Exemple de spécification

$$\forall I, \text{Precondition}(I) \implies \exists O, \text{Postcondition}(I, O)$$

| Un algorithme de tri :

- ▶ *Entrée* : un tableau t .
- ▶ *Précondition* : il existe une relation d'ordre sur les éléments du tableau.
- ▶ *Sortie* : un tableau u .
- ▶ *Postcondition* : u est trié et contient exactement les mêmes éléments que t .

Distance entre spécification et réalisation

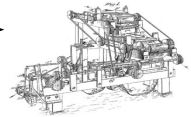
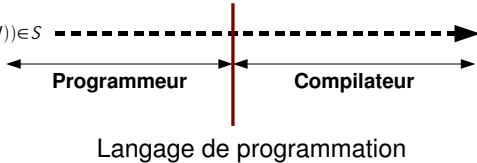
- | Ce n'est pas parce que l'on peut démontrer qu'il existe une solution à un problème que cette solution est calculable efficacement.
- | C'est au programmeur de l'explicitier à l'aide du langage de programmation.

À quel niveau de détails doit-on expliciter la solution ?

Spécification du problème

Résolution effective par la machine

$\exists F. \forall I. (I, F(I)) \in S$



- | Autrefois, il y a bien longtemps, les programmes étaient écrits sur des fiches perforées.
- | La manipulation de ces fiches était un travail de fourmi, long et fastidieux.
- | Éviter les erreurs dans la réalisation d'un programme était alors très difficile.
- | Aujourd'hui, les outils de développement logiciel ont simplifié ce processus.

Comment éviter les erreurs ?

9/9

0800 Antan started
1000 " stopped - antan ✓
1300 (032) MP-MC 2.130476415 ~~(2.130476415)~~ 4.615925059(-2)
(033) PRO 2 2.130476415
correct 2.130676415

Relays 6-2 in 033 failed special speed test
in relay 11.00 test.

Relays changed

1100 Started Cosine Tape (Sine check)
1525 Started Multi-Adder Test.

1545

Relay #70 Panel F
(moth) in relay.

First actual case of bug being found.

1630 Antan started.

1700 closed down.

Relay
2145
Relay 3370

Le premier « bug ».

Comment éviter les erreurs ?

- | Il y a différents types d'erreurs dans un programme.
- | Les erreurs de **fonctionnement** apparaissent lorsque la description du calcul contient une opération illicite qui empêche l'obtention d'un résultat :
 - ▶ "segmentation fault" : écriture dans une zone mémoire non allouée ;
 - ▶ "Uncaught exception" : un cas exceptionnel n'est pas traité.
- ⇒ Il faut s'assurer que les **opérations effectuées par la machine ont un sens**.
- | Lorsque le programme produit un résultat mais qu'il ne respecte pas la spécification, on parle d'erreur de **correction**.
- ⇒ Le calcul n'est pas le bon ! Il faut **raisonner sur le programme** pour comprendre **ce qu'il calcule**. Est-il facile de raisonner sur un programme ? Peut-on prouver qu'un programme vérifie une spécification ?

Résoudre la tension entre le « quoi ? » et le « comment ? »

- | Pour faciliter le raisonnement sur les programmes, les langages de programmation réduisent la distance entre le « quoi ? » et le « comment ? » en

*Introduisant des mécanismes calculatoires
de plus en plus en abstraits et éloignés des détails de la machine.*

- | **Abstraire**, c'est capturer ce qui est strictement utile à la programmation en mettant de côté certains détails d'implémentation non pertinents (relativement aux types de programmes écrits).
La programmation est ainsi plus sûre.

Des langages de programmation plus ou moins abstraits

- Plus un langage de programmation fournit des mécanismes calculatoires abstraits des détails de fonctionnement de la machine et plus le langage est dit de **haut-niveau**.
- Comparez les deux programmes suivants, sont-ils équivalents ?

```
let neglist = List.map (fun x → - x)
```

```
list_t neglist (list_t l) {  
  list_t result = empty_list;  
  while (l != empty_list) {  
    result = insert (- l→element, result);  
    l = l→next;  
  }  
  return result;  
}
```

Des langages de programmation plus ou moins abstraits

- Plus un langage de programmation fournit des mécanismes calculatoires abstraits des détails de fonctionnement de la machine et plus le langage est dit de **haut-niveau**.
- Comparez les deux programmes suivants, sont-ils équivalents ?

Non ! Pas du tout !

```
let neglist = List.map (fun x → - x)
```

```
list_t neglist (list_t l) {  
  list_t result = empty_list;  
  while (l != empty_list) {  
    result = insert (- l→element, result);  
    l = l→next;  
  }  
  return result;  
}
```

Exemple : s'abstraire de l'ordre des calculs

- | Langages fonctionnels :
Un programme est un ensemble de fonctions.
- | Langages logiques :
Un programme est un ensemble de formules.
- | Langages de programmation par contraintes :
Un programme est un ensemble de contraintes à satisfaire.
- | Langages orientés objets :
Un programme est un ensemble d'objets qui collaborent en communiquant par envoi de messages.

Comment décrire le calcul ?

- | Dès lors qu'ils s'attachent des opérations élémentaires de la machine qui les exécutent, les mécanismes des langages de programmation ne peuvent plus s'exprimer simplement en fonction de ces derniers.
- | Il faut se doter d'une interprétation univoque et indépendante de chaque langage de programmation : une **sémantique**.

Comment définir une sémantique ?

- | On peut donner la sémantique à un langage de plusieurs façons.
- | Une sémantique **dénotationnelle** interprète un programme dans un espace mathématique (comme une fonction par exemple).

$$\begin{aligned}\text{fact} : \quad & \mathbb{N} \rightarrow \mathbb{N} \\ & n \mapsto n!\end{aligned}$$

- | Une sémantique **opérationnelle** en donne une description calculatoire :

$$n! = 1 \times 2 \times \dots \times (n-1) \times n$$

- | En suivant les règles données par la sémantique opérationnelle d'un langage de programmation \mathcal{L} , on peut écrire un programme, appelé **interprète**, qui permet d'exécuter tout programme écrit en \mathcal{L} .

Un exemple de sémantique opérationnelle

- | Un langage de programmation est défini ainsi :

- ▶ Un programme est une liste d'instructions.
- ▶ Une instruction est l'une des deux formes suivantes :
 - ▶ Avance N où N est un entier représentant un nombre de pixel.
 - ▶ Tourne N où N est un entier représentant un angle en degré.

- | On suppose un environnement d'exécution constitué d'un plan fini à deux dimensions, d'une position courante et d'une orientation courante.
- | La sémantique opérationnelle pourrait alors être décrite ainsi :

- ▶ Pour chaque instruction i d'un programme P :
 - ▶ Si $i \equiv \text{"Avance } N"$ alors tracer une ligne de longueur N depuis la position courante et en suivant la direction courante.
 - ▶ Si $i \equiv \text{"Tourne } N"$ alors la direction courante est augmentée de l'angle N .

⇒ Comment écrire un interprète pour ce langage ?

Qu'est-ce qui différencie un langage d'un autre ?

- | Il y a une infinité de façon de représenter un calcul.
 - | Dans le cas précédent, un calcul est une succession de lignes tracées à l'écran.
 - | En C, un calcul est une transformation de l'état de la machine.
 - | Dans le cas d'un mélange chimique, un calcul est un ensemble de réactions.
- ⇒ Ce sont autant de **modèles de calcul** différents.

Une machine abstraite pour chaque modèle de calcul

- | Une **machine abstraite** décrit l'environnement et les règles d'évaluation d'un modèle de calcul.
- | Très peu de modèles possèdent une réalisation physique de leur machine abstraite (à l'exception du modèle de Von Neumann).
- | On peut cependant **émuler** une machine abstraite à l'aide d'un autre programme.
- | Un tel programme est appelé **machine virtuelle**.

Exemples de machines virtuelles

- | La *Java Virtual Machine* (JVM) est une machine virtuelle implémentée par le programme `java`. Le programme `ocamlrun` est la machine virtuelle du langage OCaml.
- | Le programme *Qemu* émule un ordinateur IBM PC.
- | On peut écrire un programme qui émule une machine de Turing.

Une limitation gênante, un compilateur pour la contrer

- | L'émulation d'une machine abstraite ou l'utilisation d'un interprète introduisent une *inévitabile inefficacité* : pour chaque opération X de la machine abstraite, il est nécessaire de décoder sa représentation et de l'émuler au fur et à mesure par des instructions de la machine réelle. Il y a donc une couche supplémentaire de traduction.
- | Un compilateur répond à l'objectif de limitation de ces couches d'interprétation en *traduisant un programme écrit dans un langage \mathcal{L} en un programme équivalent écrit dans un langage \mathcal{L}* et qui possède une réalisation efficace de sa machine abstraite.

Exemples de compilation

- | Le langage C se compile en code assembleur *i386* qui est interprété (très) efficacement par les processeurs d'architecture *Intel*.
- | Le langage Java se compile vers un langage appelé code-octet Java qui est interprété par la machine virtuelle Java.
- | Python est compilé vers un sous-ensemble de Python qui est interprété par la machine virtuelle de Python.

La spécification d'un compilateur

- | Un compilateur attend des programmes écrits dans un certain **langage source** et qui ont un sens.
- | La précondition d'un compilateur est donc une certaine propriété de bonne formation des programmes d'entrée (être syntaxiquement corrects, être bien typés, ...).
- | La sortie d'un compilateur est un programme écrit dans le *langage cible*.
- | On s'attend d'une part à ce que ce programme ait un sens mais, et c'est bien plus fort, à ce qu'il **ait un sens équivalent** au programme d'entrée.
- | En résumé, la propriété fonctionnelle essentielle d'un compilateur est donc d'être **une traduction préservant la sémantique** des programmes.

Quelques mises au point. . .

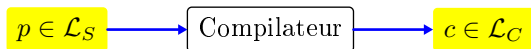
- | Un compilateur est une fonction qui traduit un programme écrit dans un langage source en un programme écrit dans un langage cible.
- | Un interprète pour un langage L est une fonction qui évalue un programme écrit dans le langage L .
- | En pratique, ces outils s'utilisent de multiples façons :
 - ▶ Le mode *batch* qui consiste à intégrer la fonction de compilation ou d'interprétation dans un programme et à l'utiliser sur des fichiers, appelés unités de compilation.
 - ▶ Le mode interactif qui consiste à intégrer la fonction de compilation et/ou d'interprétation dans une boucle interactive de façon à compiler et/ou interpréter les entrées de l'utilisateur au fur et à mesure de leur définition.

Exemples d'outils d'interprétation et de compilation

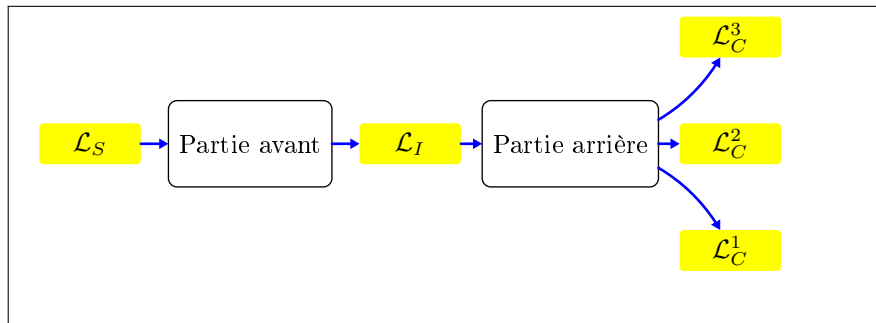
- | Le programme `python` est un interprète en mode interactif quand il est utilisé sans argument, en mode *batch* si on lui donne un fichier `py`.
- | Le programme `ocaml` est un compilateur vers la machine virtuelle `ocaml` et un interprète pour cette machine virtuelle dans une boucle interactive.
- | Les programmes `gcc`, `ocamlc`, `ocamlopt`, `javac` sont des compilateurs en mode *batch*.

Quels sont les techniques d'implémentation utilisées en compilation ?

Architecture d'un compilateur (de loin)

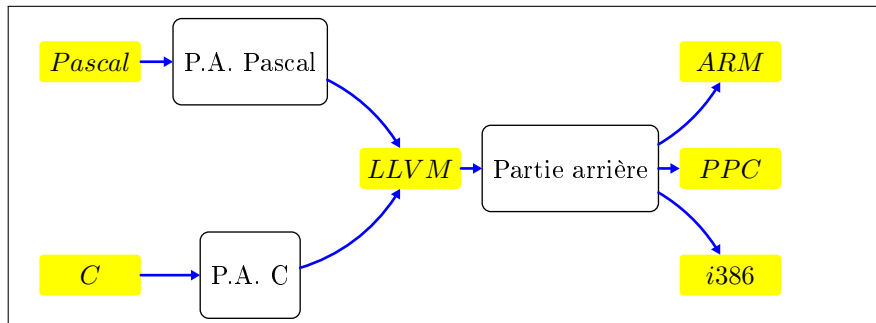


Architecture d'un compilateur (d'un peu plus près)



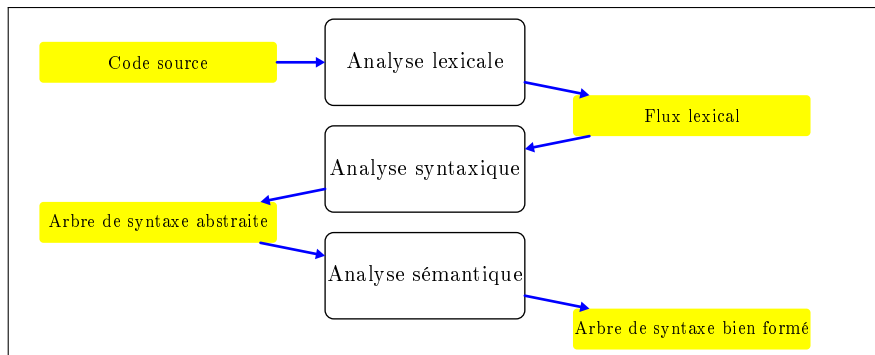
- Une architecture très courante consiste à séparer le compilateur en deux sous-compilateurs.
- La **partie avant** (*front-end*) traduit un programme du langage source \mathcal{L}_S en un programme d'un langage intermédiaire \mathcal{L}_I .
- Ce programme de \mathcal{L}_I est ensuite traduit par la **partie arrière** du compilateur en programmes de langages cibles différents.

Architecture d'un compilateur (d'un peu plus près)



- On peut ainsi réutiliser la partie arrière pour factoriser la conversion vers des cibles différentes.
- Le choix du langage intermédiaire est crucial.

L'architecture d'un compilateur : la partie avant



- Le cours de cette année porte essentiellement sur la partie avant.

Un micro-compilateur dans une boucle interactive

- | Pour comprendre les différentes phases de la compilation, nous allons implémenter un micro-compilateur pour un langage de calcul arithmétique, que nous appellerons Marthe.

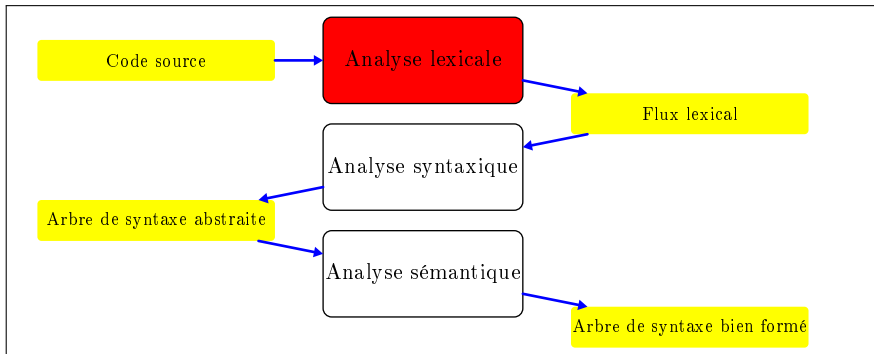
Structure d'une boucle interactive

Read - Eval - Print Loop

- | On structure traditionnellement les boucles interactives à l'aide de quatre fonctions :
 - ▶ “Read” : on lit une chaîne représentant un programme dans un langage L .
 - ▶ “Eval” : on évalue cette chaîne si elle a du sens.
 - ▶ “Print” : on affiche le résultat.
 - ▶ “Loop” : on revient à la première étape.
- | La fonction “Eval” peut se réaliser de différente façon :
 - ▶ grâce à un interprète du langage L ;
 - ▶ grâce à un compilateur du langage L vers un langage L' pour lequel on a un interprète.
 - ▶ avant l'exécution du programme, un *analyseur statique* peut vérifier qu'il va s'exécuter sans erreur.

Boucle interactive en O'Caml

```
let loop read eval print =  
  let rec aux () =  
    let entry = read () in  
    let result = eval entry in  
    print result ;  
    aux ()  
  in  
  aux ()
```



Un micro-compileur : langage source

- | Traditionnellement, le code source d'un programme est écrit dans un éditeur de texte et sauvegarder sous cette forme. Dans une boucle interactive, il peut aussi être saisi à l'aide d'un terminal.
- | Dans tous les cas, cette entrée textuelle est une séquence de caractères (ASCII ou UNICODE).
- | Voici quelques exemples de programmes que pourrait écrire un programmeur à destination de notre micro-compileur de Marthe :

1. `"7 * 3 + 7 * 3"`
2. `"sum (i, 1, 10, i * i)"`
3. `"sum (j, 1, 5, i + i / 2)"`
4. `"-3 / 0 + 1"`

- | Peut-on donner un sens à tous ces programmes ?

Un micro-compileur : analyse lexicale

On ne veut pas différencier les programmes suivants :

1. `"7 * 3 + 7 * 3"`

2. `"7 * 3 + 7 * 3"`

3. `" 7 *3 +
 7 * 3"`

4. `"(* Ceci est un commentaire. *) 7 *3 + 7 * 3"`

Un micro-compileur : analyse lexicale

- | L'**analyse lexicale** traduit une chaîne de caractères en un flux de lexèmes (*tokens* en anglais) : ce sont les unités lexicales qui ont un sens du point de vue du langage de programmation et qui interviennent dans la spécification de sa grammaire.
- | Les programmes précédents sont tous transformés en :

INT(7) ; STAR ; INT(3) ; PLUS ; INT(7) ; STAR ; INT(3)

- ▶ "INT(*k*)" signifie "un entier valeur *k*".
- ▶ "STAR" représente le symbole "*".
- ▶ "PLUS" représente le symbole "+".
- ▶ Les espaces, sauts de ligne et commentaires, quelque soit leur nombre, ont servi de séparateurs entre ces unités lexicales.

Un analyseur lexical en O'Caml

```
type token =  
  | Int of int  
  | Id of string  
  | Sum  
  | Plus  
  | Star  
  | Lparen  
  | Rparen  
  | Comma  
  
exception LexingError of string
```

```
let lexer s =  
  let rec number start i =  
    let return () =  
      (int_of_string (String.sub s start (i - start)), i)  
    in  
    if i ≥ String.length s then return () else match s.[i] with  
    | '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' →  
      number start (i + 1)  
    | _ →  
      return ()  
  in  
  let rec identifier start i =  
    let return () = (String.sub s start (i - start), i) in  
    if i ≥ String.length s then return () else match s.[i] with  
    | c when c ≥ 'a' ∧ c ≤ 'z' →  
      identifier start (i + 1)  
    | _ →  
      return ()  
  in  
  let rec aux i =  
    if i ≥ String.length s then [EOF] else match s.[i] with  
    | '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' →  
      let (n, i) = number i i in  
      Int n :: aux i  
    | ' ' → aux (i + 1)  
    | 'x' → Star :: aux (i + 1)  
    | '+' → Plus :: aux (i + 1)  
    | '(' → Lparen :: aux (i + 1)  
    | ')' → Rparen :: aux (i + 1)  
    | c when c ≥ 'a' ∧ c ≤ 'z' →  
      let (s, i) = identifier i i in  
      (if s = "sum" then Sum else Id s) :: aux i  
    | ',' → Comma :: aux (i + 1)  
    | _ →  
      raise (LexingError "Invalid character")  
  in  
  aux 0
```

Un micro-compileur : analyse lexicale

- | L'analyse lexicale est implémentée à l'aide d'un *automate fini* (la plupart du temps).
- | On décrit chaque sorte de lexème à l'aide d'une **expression régulière**.
- | Exemples :
 - ▶ $[0-9]^+$: les entiers naturels en base 10.
 - ▶ $[a-z]^+$: les identifiants formés de lettres minuscules.
 - ▶ $(A|B)[a-z]^*$: les identifiants qui commencent par un 'A' ou un 'B'.
- | L'automate correspondant à cette expression régulière est calculé.
- | On parcourt la chaîne de caractère du programme source.
- | À partir du point courant de cette chaîne, on détermine quel est l'automate qui reconnaît **la plus longue sous-chaîne** pour savoir quel lexème produire.

Un micro-compileur : analyse lexicale

- | Le programme qui calcule les automates est un **générateur d'analyseur lexical**.
- | En OCaml, on utilise `ocamllex`. En C, on utilise `flex`.
- | Ces programmes sont des **générateurs de code** : il transforme une spécification d'analyseur lexical en un programme (OCaml ou C par exemple).
- | Leur utilisation sera l'objet du prochain TD.

Un micro-compileur : analyse lexicale

```
{
  type token = Int of int / Plus / Star
}

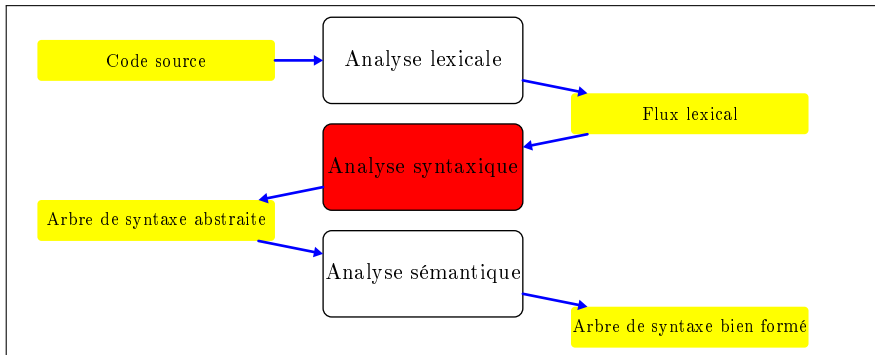
rule token = parse
  / ['0'-'9']+ { Int (int_of_string (Lexing.lexeme lexbuf)) }
  / "+" { Plus }
  / "x" { Star }
  / " " { token lexbuf }
  / _ { failwith "Unknown character." }

{
  let print = function
    / Int x    Printf.printf "Int(%d)" x
    / Plus    Printf.printf "Plus"
    / Star    Printf.printf "Star"

  let lexbuf = Lexing.from_string Sys.argv.(1)

  let rec loop () =
    try
      print (token lexbuf);
      Printf.printf "; ";
      loop ()
    with _   ()

  let _ = loop ()
}
```



Un micro-compileur : analyse syntaxique

- | Un flux de lexèmes a une **structure linéaire** qui n'est pas pratique pour interpréter le programme.
- | À première vue, écrire un programme qui interprète correctement :

`INT(7) ; STAR ; INT(3) ; PLUS ; INT(7) ; STAR ; INT(3)`

... n'est pas aisé puisque l'on ne peut pas prendre une décision locale.

- | En effet, il serait erroné d'effectuer l'addition avant d'avoir calculé le résultat de la multiplication $7 * 3$, à gauche et à droite.

Micro-compileur : analyse syntaxique

- A priori, certaines séquences de lexèmes n'ont pas de structure associée :

1. "7 * 3 + "
2. "7 ** 3"
3. "(1 + 3))"

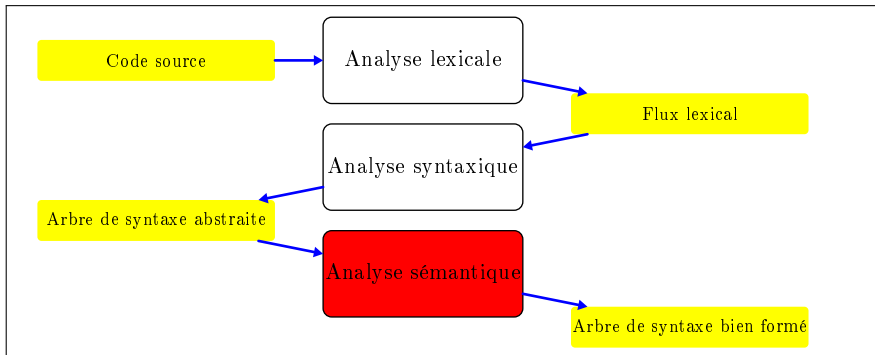
- La grammaire du langage spécifie les séquences syntaxiquement correctes.
- Dans le cas de Marthe, la syntaxe des expressions pourrait être :

```
e ::= INT  
    | e PLUS e  
    | e STAR e  
    | SUM LPAREN ID COMMA e COMMA e COMMA e RPAREN
```

- La notation utilisée ici s'appelle **BNF**, pour Backus-Naur Form.

Micro-compileur : analyse syntaxique

- | Le rôle de l'analyse syntaxique est double :
 1. Rejeter les programmes incorrects syntaxiquement.
 2. Produire un arbre de syntaxe abstraite dans le cas contraire.
 - | Les **générateurs d'analyseurs syntaxiques** transforment une spécification de grammaire en un programme exécutable qui réalise les deux opérations précédentes.
- ⇒ Nous étudierons les algorithmes sur lesquels ils s'appuient.



Un analyseur syntaxique en O'Caml

```
let parse tokens =
  let (accept, current, next) =
    let tokensstream = ref tokens in
    let next () =
      match !tokensstream with
      | [] → raise (ParseError ("No more token", EOF))
      | tok :: tokens →
          tokensstream := tokens
    in
    let current () =
      match !tokensstream with
      | [] → assert false
      | tok :: _ →
          tok
    in
    let accept token =
      if (current () ≠ token) then
        raise (ParseError ("Unexpected token", token));
      next ()
    in
    (accept, current, next)
  in

  let rec phrase () =
    let e = expression () in
    accept EOF ;
    e
```

```
and expression () =
  let e = term () in
  match current () with
  | EOF → e
  | Plus → next () ; EPlus (e, expression ())
  | token → e

and term () =
  let t = factor () in
  match current () with
  | Star → next () ; EMult (t, term ())
  | token → t

and factor () =
  match current () with
  | Lparen →
      next () ; let e = expression () in accept Rparen ; e
  | Sum →
      next () ;
      accept Lparen ;
      let id =
        match current () with
        | Id s → next () ; s
        | token →
            raise (ParseError ("Expecting an identifier",
                                token))
      in
      accept Comma ;
      let start = expression () in accept Comma ;
      let stop = expression () in accept Comma ;
      let body = expression () in accept Rparen ;
      ESum (id, start, stop, body)
  | Id x → next () ; EVar x
  | Int x → next () ; EInt x
  | token →
      raise (ParseError ("Unexpected token", token))
  in
  phrase ()
```

Micro-compileur : analyse sémantique

- | On peut donner une sémantique à Marthe à l'aide de phrases françaises :
 - ▶ Pour calculer $e_1 + e_2$, calculer e_1 , calculer e_2 et faire la somme des deux résultats.
 - ▶ Pour calculer $e_1 * e_2$, calculer e_1 , calculer e_2 et faire le produit des deux résultats.
 - ▶ Pour calculer $sum(i, e1, e2, e3)$, calculer la valeur *start* de $e1$, calculer la valeur *stop* de $e2$, faire la somme de toutes valeurs de $e3$ pour i valant tous les entiers de *start* à *stop*.
 - | La sémantique est ici opérationnelle et s'intéresse uniquement à la valeur issue de l'évaluation de l'expression.
- ⇒ Nous utiliserons un **formalisme** plus clair et plus concis pour définir les sémantiques opérationnelles des langages de programmation à base de règles d'inférence.

Micro-compileur : analyse sémantique

- | Malgré une syntaxe correcte, certains programmes n'ont pas de sens.
- | Les deux premiers des programmes suivants calculent respectivement 42 et $\sum_{i=1}^{10} i^2$ tandis que les deux derniers n'ont pas d'évaluation.

1. "7 * 3 + 7 * 3"
2. "sum (i, 1, 10, i * i)"
3. "sum (j, 1, 5, i + i / 2)"
4. "-3 / 0 + 1"

- | On aimerait rejeter le programme numéro 3 parce qu'il fait référence à une variable i indéfinie. On aimerait peut-être aussi alerter le programmeur de la non-utilisation de la variable j .
 - | On aimerait rejeter le programme numéro 4 parce qu'il divise par zéro.
- ⇒ Peut-on formuler une règle générale (calculable, menant à un algorithme) pour décider le rejet ou l'acceptation d'un programme ?

Micro-compileur : analyse sémantique

I Règle 1 : “Pas de référence à des variables indéfinies”

Soient un arbre de syntaxe P et \mathcal{V} , la liste des variables définies pour P .

On suit un parcours préfixe de l'arbre et on applique les cas suivants :

- ▶ *Si la racine de P est un nœud sum alors son premier sous-arbre est une variable i , la rajouter dans la liste \mathcal{V} et continuer le parcours dans le dernier sous-arbre.*
- ▶ *Si la racine de P est une variable i , vérifier que i est dans \mathcal{V} .*

Micro-compileur : analyse sémantique

| Essai de règle 2 : "Pas de division par zéro"

Soient un arbre de syntaxe P . On suit un parcours préfixe de l'arbre et on applique les cas suivants :

- ▶ *Si la racine de P est un nœud "division" alors si le second sous-arbre s'évalue en zéro, on doit rejeter le programme.*

- | Évaluer le programme que l'on compile n'est pas vraiment envisageable. D'abord, parce que ce n'est pas le rôle d'un compilateur mais celui d'un interprète. Ensuite, un programme écrit dans un langage généraliste peut ne pas terminer : si le programme ne termine pas alors le compilateur risque de ne jamais terminer lui non plus !

Micro-compilateur : analyse sémantique

| Nouvel essai de règle 2 : “Pas de division par zéro”

Soient un arbre de syntaxe P . On suit un parcours préfixe de l'arbre et on applique les cas suivants :

- ▶ *Si la racine de P est un nœud “division” et si le second sous-arbre est la constante 0 alors on rejette le programme.*

- | Ce n'est toujours pas satisfaisant !
- | Le compilateur ne va pas rejeter le programme “ $1/(0 + 0)$ ” puisque “ $0 + 0$ ” n'est pas syntaxiquement égal à 0.
- | Si un programme est accepté, alors cela ne signifie pas qu'il ne divisera pas par 0.

Micro-compileur : analyse sémantique

- | Existe-t-il une règle qui accepte seulement des programmes qui ne divisent pas par zéro ?
- ⇒ On peut bannir la division ! Une autre idée ?
- | Existe-t-il une règle qui accepte **uniquement les** programmes qui ne divisent pas par zéro (sans l'évaluer) ?

Micro-compileur : analyse sémantique

- | Le rôle de l'**analyse sémantique** est d'appliquer des règles pour fournir des garanties sur l'exécution du programme compilé.
- | Dans notre exemple, la règle numéro 1 garantit qu'au cours du calcul, on fera toujours référence à des variables connues. Dans le cas de notre langage, ceci suffit pour s'assurer que tous les calculs aboutiront à un résultat entier si aucune division par zéro n'a lieu.
- | Nous nous concentrerons sur des algorithmes (de typage) permettant d'obtenir ce type de propriétés de sûreté de l'évaluation.

Interprète

- À l'aide de la sémantique opérationnelle, écrire un interprète sur les arbres de syntaxes abstraites est très simple :

```
let rec eval variables = function
  / Int x      x
  / Var x      List.assoc x variables
  / Plus (e1, e2)  eval variables e1 + eval variables e2
  / Sum (x, e1, e2, e3)
    let start = eval variables e1 in
    let stop = eval variables e2 in
    let accu = ref 0 in
    for i = start to stop do
      accu := !accu + eval ((x, i) :: variables) e3
    done ;
    !accu
```

- On utilise l'analyse de motif (*pattern matching*) d'OCaml et la récursion pour écrire une fonction définie par induction sur l'arbre de syntaxe abstraite.
 - La machine abstraite du langage est formée de la liste des valeurs courantes des variables et du terme à évaluer.
- ⇒ Peut-on supprimer cette couche d'interprétation en compilant le programme vers un modèle d'exécution plus rapide ?

Micro-compileur : partie arrière

- | Soit le langage de programmation Asm dans lequel un programme est une suite d'instructions de la forme suivante :
 - ▶ **"pushi n "** où n est un entier.
 - ▶ **"pushv x "** où x est une variable.
 - ▶ **"add", "sub", "mul", "div"**.
 - ▶ **"for $i = start$ to $stop$ do $i_1; \dots i_n$; done"**
- | La machine abstraite de ce langage est formé d'une pile contenant entiers et noms de variable, d'une liste d'instructions à exécuter et d'un pointeur indiquant l'instruction en cours d'exécution.
- | On peut traduire un arbre de syntaxe de Marthe en un arbre de syntaxe de Asm : il suffit d'expliciter l'ordre dans lequel les calculs sont effectués et on stocke les résultats temporaires sur la pile.

Micro-compileur : partie arrière

- | Voici quelques exemples de traductions :
 - ▶ “ $2 + 40$ ” \Rightarrow “**pushi 2 ; pushi 40 ; add**”.
 - ▶ “ $3 * 7 + 3 * 7$ ” \Rightarrow “**pushi 3 ; pushi 7 ; mul ; pushi 3 ; pushi 7 ; mul ; add**”
- | La structure du programme est de nouveau linéaire !
- | Mais cette fois-ci, le programme est très facile à évaluer...

Micro-compilateur : partie arrière

- | On peut écrire un interprète pour Asm.
- | On peut aussi compiler Asm vers un autre langage avec un modèle d'exécution plus efficace et ainsi de suite.

Pourquoi étudier la compilation ?

Pourquoi étudier la compilation ?

- | La probabilité que l'on vous demande d'écrire un compilateur au cours de votre vie professionnelle est faible.¹
- | Pourquoi donc étudier des algorithmes spécialisés à ce domaine ou la théorie des langages ?

1. Cependant, les langages dédiés à un domaine spécifique (*Domain Specific Languages*, DSL) ont connu un important essor ces dernières années et l'implantation de ces derniers nécessitent l'écriture de compilateurs. . .

Pour devenir de meilleurs programmeurs

- | La compréhension des mécanismes fondamentaux des langages de programmation attire votre regard sur les programmes que vous écrivez.
 - | Un langage de programmation est une somme de mécanismes calculatoires.
 - | Ces mécanismes sont partagés entre les différents langages.
 - | Connaître ces mécanismes, c'est se faciliter l'apprentissage de nouveaux langages de programmation.
 - | Les comprendre en profondeur rend possible un raisonnement rigoureux sur les programmes.
- ⇒ Le cours de compilation est avant tout un cours de **programmation**.

Pour faire croître sa boîte à outils

- | Si l'écriture de compilateur n'est pas une activité quotidienne pour la plupart des développeurs, l'utilisation et l'écriture d'outils d'analyse, de gestion et de génération de code sont très courantes.
- ⇒ Une partie des algorithmes de la compilation sont réutilisables dans ce contexte.
- | Beaucoup de problèmes se comprennent mieux quand on s'est donné un langage pour les spécifier.
- | Les langages sont partout : un format de fichier est un langage, une requête à un système s'écrit dans un langage, les traces des protocoles réseaux forment elles-aussi un langage. . .
- ⇒ La théorie des langages va vous donner les outils scientifiques pour les spécifier et implémenter rigoureusement.

Pour devenir de meilleurs informaticiens

- | La compilation est un **sujet transversal** de l'informatique.
- | Pour écrire un compilateur, on sollicite ses connaissances :
 - ▶ des automates ;
 - ▶ des algorithmes travaillant sur les arbres et sur les graphes ;
 - ▶ des architectures des ordinateurs ;
 - ▶ de théorie des langages ;
 - ▶ de génie logiciel.

La compilation dans le cursus de Paris Diderot

- | L3 : Introduction à la compilation, Machines virtuelles, Programmation Fonctionnelle.
- | M1 : Compilation, Sémantique, Génie logiciel, P.O. avancée, Programmation Fonctionnelle Avancée.
- | M2 (LP) : Compilation avancée, P.O concepts avancés, Programmation Comparée.
- | M2 (MPRI) : Tous les cours liés à la sémantique des langages de programmation.

Synthèse

Conclusion de ce premier cours introductif

- | Le compilateur permet à la programmation de s'abstraire des détails de la machine en traduisant un langage source, de haut-niveau, en un langage cible, de plus bas niveau et qui possède un modèle d'exécution plus efficace.
- | Un compilateur est généralement la composée de plusieurs traductions.
- | Nous nous intéresserons à l'analyse lexicale et syntaxique au prochain cours.

Plan du cours

- | Analyse syntaxique du cours 2 au cours 6 :
 - ▶ Cours 2 : Introduction
 - ▶ Cours 3 : Analyse descendante
 - ▶ Cours 4 et 5 : Analyse ascendante
 - ▶ Cours 6 : Analyse syntaxique en pratique et révision
 - ▶ Partie 1 du projet : Un analyseur syntaxique à partir d'une spécification.
- | Sémantique :
 - ▶ Cours 7 : De la syntaxe à la sémantique
 - ▶ Cours 8 : La liaison des variables
 - ▶ Cours 9 : L'état
 - ▶ Cours 10 et 11 : Les procédures
 - ▶ Cours 12 : Les fonctions de première classe
 - ▶ Cours 13 : Compilation des fonctions de première classe
 - ▶ Partie 2 du projet : Un interprète et un typeur pour le langage de la partie 1.

Fonctionnement du cours

- | Le cours a lieu le vendredi de 10h30 à 12h30 en amphi 6C.

Venez en cours !

- | Les travaux dirigés :
 - ▶ Le lundi de 9h30 à 11h30, Daniele Varacca en 2031.
 - ▶ Le lundi de 14h30 à 16h30, Mihaela Sighireanu en 2031.
 - ▶ Le mardi de 10h30 à 12h30, Ahmed Bouajjani en 2032.
- | Modalités d'évaluation :
 - ▶ 1ère session : 40% projet + 60% examen
 - ▶ 2ème session : 40% projet + 60% examen
 - ▶ Le projet est **obligatoire**.

Comment travailler le cours ?

I Connaître le cours :

- ▶ La page du cours, contenant les ressources comme les transparents, les programmes écrits en cours ou encore les références bibliographiques :
<http://www.pps.jussieu.fr/~yrg/compl>

I S'exercer :

- ▶ Les travaux dirigés.
- ▶ Des exercices seront donnés dans les notes de cours.
- ▶ Des exercices de programmation, non obligatoires, vous seront proposés.

QCM : Question 1

Un compilateur a pour rôle :

- a. de traduire un programme en assembleur.
- b. de vérifier qu'un programme termine.
- c. de traduire un programme dans langage dans un autre langage.
- d. d'évaluer n'importe quel programme d'un certain langage.

QCM : Question 2

Un langage de programmation de haut-niveau :

- a. est nécessairement interprété.
- b. produit, une fois compilé, des programmes systématiquement plus lents que les programmes écrits dans des langages de bas-niveau.
- c. aide à éviter certaines erreurs de programmation.
- d. complique le raisonnement sur les programmes.

QCM : Question 3

La partie avant d'un compilateur :

- a. comprend une phase d'analyse lexicale.
- b. comprend une phase d'analyse syntaxique.
- c. comprend une phase d'analyse sémantique.
- d. comprend une phase de génération de code.
- e. doit être réécrite pour chaque processeur.
- f. doit être réécrite pour chaque langage.

QCM : Question 4

Laquelle de ces structures de données est la plus pratique pour représenter et évaluer une expression arithmétique :

- a. une chaîne de caractère en notation infixe sans parenthésage explicite.
- b. une chaîne de caractère en notation infixe avec parenthésage explicite.
- c. une chaîne de caractère en notation postfixe.
- d. un arbre de syntaxe abstraite.

QCM : Question 5

On peut écrire un algorithme qui a pour entrée le code source d'un programme P et détermine :

- a. si P termine.
- b. si P fera "*segmentation fault*".
- c. si P fera une division par zéro.
- d. si P peut faire une division par zéro.

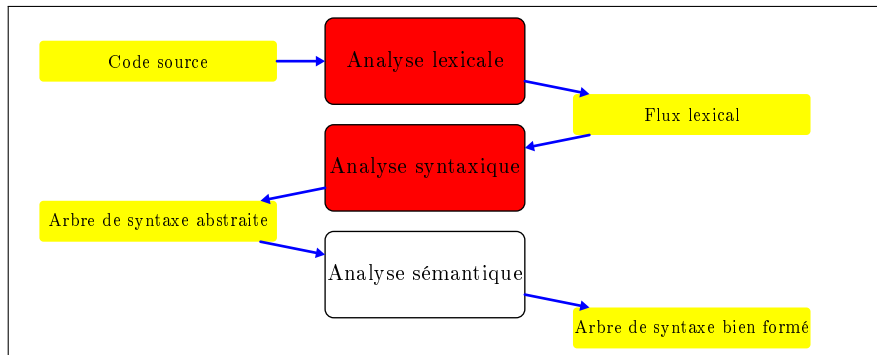
INTRODUCTION À LA COMPILATION

Cours 2 : Analyse lexicale et syntaxique

Yann Régis-Gianas
yrg@pps.univ-paris-diderot.fr

PPS - Université Denis Diderot – Paris 7

Rappel du dernier cours



Première partie du cours :

L'analyse lexicale et l'analyse syntaxique dans la partie avant d'un compilateur.

Vue d'ensemble

Grandes lignes

- | L'analyse lexicale traduit un flux de caractères en un flux de lexèmes.
 - | L'analyse syntaxique traduit un flux de lexèmes en un arbre de syntaxe abstraite.
- ⇒ Dans les deux cas, il s'agit d'exhiber la structure implicite d'un flux.
- | Cette structure nous servira à donner un sens aux programmes.
- ⇒ Pour le moment, nous nous focalisons sur les problèmes suivants :
1. Comment spécifier la structure de tous les programmes d'un langage \mathcal{L} ?
 2. Comment reconnaître la structure d'un programme et la reconstruire ?

Décrire une syntaxe

| Pour voir un programme comme un arbre, on doit répondre à ces questions :

- | Quelles sont ses feuilles ? – Les symboles **terminaux**.
⇒ les mots-clés du langage, sa « ponctuation », ses atomes de sens.
- | Quels sont ses nœuds ? – Les symboles **non terminaux**.
⇒ ses **catégories syntaxiques** (expressions, déclarations, types, ...).

Décrire une syntaxe : exemple du langage MARTHA

| **Terminaux** :

- ▶ Mots-clés : **sum**, **from**, **to**, **do**, **def**.
- ▶ Identifiants : x, y, z, foo, bar, ...
- ▶ Constantes entières : 0, 1, 2, 3, ...
- ▶ Symboles : '+', '*', '/', '-', '(', ')', ',', '=', ...

| **Non terminaux** :

- ▶ Expression : les expressions arithmétiques formées :
 - ▶ par application des symboles d'opérateurs appliqués à des sous-expressions, en respectant les priorités entre ces symboles ;
 - ▶ par application de l'opérateur **sum** suivi d'un identifiant, du mot-clé **from**, d'une expression, du mot-clé **to**, d'une expression, du mot-clé **do** et d'une expression ;
 - ▶ par application d'une fonction à une liste d'expressions, séparées par des virgules et entourée de parenthèses.
- ▶ Définition de fonctions : le mot-clé **def** suivi d'un identifiant, d'une liste d'identifiants entre parenthèses et séparés par des virgules, du symbole '=' et d'une expression.
- ▶ Programme : une liste de définitions de fonction suivie d'une expression.

⇒ Une description peu concise ... et incomplète !

(Quelles sont les priorités respectives des symboles ? Comment les décrire ?)

Grammaires

Une **grammaire** (**génération**, **syntagmatique**, **formelle**, ...) est un quadruplet N, T, R, S où :

- | N et T sont des ensembles de symboles disjoints.
- | R est un ensemble de paires P, Q avec $P \in N \cup T^+$ et $Q \in N \cup T^*$ ¹
- | Il existe un symbole $S \in N$.

Les règles de la grammaire P, Q de R , sont en général notés « $P \rightarrow Q$ ».

- | N est l'ensemble des symboles non-terminaux.
- | T est l'ensemble des symboles terminaux.
- | R est l'ensemble des règles.
- | S est le symbole d'entrée.

1. A^+ représente l'ensemble des séquences non vides d'éléments de A tandis que A^* représente l'ensemble des séquences vides d'éléments de A .

La grammaire du langage MARTHA

- | $Keywords \equiv \{\text{sum, from, to, do, def}\}$
- | $f, x \in Identifiers \equiv \{a, \dots, z\}^+ \setminus Keywords^2$
- | $i \in Integers \equiv \{0, \dots, 9\}^+$
- | $T \equiv \{'+', '*', '/', '- ', '(', ')', ', ', '= '\} \cup Keywords \cup Identifiers$
- | $N \equiv$
 $\{Expression, Factor, Term, SExpressions\} \cup$
 $\{SIdentifiers, Definition, Definitions, Program\}$
- | R est :

<i>Expression</i>	\rightarrow	<i>Expression</i> '+' <i>Term</i>
<i>Expression</i>	\rightarrow	<i>Expression</i> '- ' <i>Term</i>
<i>Expression</i>	\rightarrow	<i>Term</i>
<i>Term</i>	\rightarrow	<i>Term</i> '* ' <i>Factor</i>
<i>Term</i>	\rightarrow	<i>Term</i> '/ ' <i>Factor</i>
<i>Term</i>	\rightarrow	<i>Factor</i>
<i>Factor</i>	\rightarrow	<i>i</i>
<i>Factor</i>	\rightarrow	<i>x</i>
<i>Factor</i>	\rightarrow	<i>f</i> '(' <i>SExpressions</i> ')'
<i>Factor</i>	\rightarrow	'(' <i>Expression</i> ')'

2. f, x sont des "méta-variables" qui dénotent des éléments pris dans l'ensemble *Identifiers*.

La grammaire du langage MARTHA (suite)

<i>SExpressions</i>	→	<i>Expression</i>
<i>SExpressions</i>	→	<i>Expression</i> ' , ' <i>SExpressions</i>
<i>SIdentifiers</i>	→	<i>x</i>
<i>SIdentifiers</i>	→	<i>x</i> ' , ' <i>SIdentifiers</i>
<i>Definition</i>	→	def <i>f</i> ' (' <i>SIdentifiers</i> ') ' '=' <i>Expression</i>
<i>Definitions</i>	→	ε
<i>Definitions</i>	→	<i>Definition Definitions</i>
<i>Program</i>	→	<i>Definitions Expression</i>

La grammaire du langage MARTHA, écriture condensée

<i>Expression</i>	\rightarrow	<i>Expression</i> '+' <i>Term</i> <i>Expression</i> '-' <i>Term</i> <i>Term</i>
<i>Term</i>	\rightarrow	<i>Term</i> '*' <i>Factor</i> <i>Term</i> '/' <i>Factor</i> <i>Factor</i>
<i>Factor</i>	\rightarrow	<i>i</i> <i>x</i> <i>f</i> '(' <i>SExpressions</i> ')' '(' <i>Expression</i> ')'
<i>SExpressions</i>	\rightarrow	<i>Expression</i> <i>Expression</i> ',' <i>SExpressions</i>
<i>SIdentifiers</i>	\rightarrow	<i>x</i> <i>x</i> ',' <i>SIdentifiers</i>
<i>Definition</i>	\rightarrow	def <i>f</i> '(' <i>SIdentifiers</i> ')' '=' <i>Expression</i>
<i>Definitions</i>	\rightarrow	ϵ <i>Definition</i> <i>Definitions</i>
<i>Program</i>	\rightarrow	<i>Definitions</i> <i>Expression</i>

- | On peut déduire *T* et *N* par simple observation des règles.
- | On utilise le symbole '|' pour dénoter une alternative dans la grammaire.

La grammaire du langage MARTHA

Exercice

- ⇒ Modifiez la grammaire pour accepter la définition et l'utilisation de fonctions d'arité nulle.³

3. C'est-à-dire dont le nombre d'arguments formels vaut zéro.

Grammaires génératives en OCAML

- On peut donner un type OCAML à ces objets :

```
type ('t, 'n) symbol =  
  | Terminal of 't  
  | NonTerminal of 'n
```

```
type ('t, 'n) lhs =  
  ('t, 'n) symbol  $\times$  ('t, 'n) symbol list
```

```
type ('t, 'n) rhs =  
  ('t, 'n) symbol list
```

```
type ('t, 'n) rule =  
  ('t, 'n) lhs  $\times$  ('t, 'n) rhs
```

```
type ('t, 'n) grammar =  
  ('t, 'n) rule list
```

Grammaire de MARTHA en OCAML

```
type ('t, 'n) rulealternative = 'n  $\rightarrow$  ('t, 'n) rule list

(* ('t, 'n) rhs  $\rightarrow$  ('t, 'n) rule_alternative *)
let (!) rhs =
  fun lhs  $\rightarrow$  [((NonTerminal lhs, []), rhs)]

(* ('t, 'n) rule_alternative  $\rightarrow$  ('t, 'n) rhs *)
(*  $\rightarrow$  ('t, 'n) rule_alternative *)
let (*|) rhs1 rhs2 =
  fun lhs  $\rightarrow$  (rhs1 lhs @ (! rhs2) lhs)

(* ('t, 'n) rule_alternative  $\rightarrow$  'n  $\rightarrow$  ('t, 'n) rule list *)
let ( $\longrightarrow$ ) lhs rhs = rhs lhs
```

Quelques notations.

Terminaux et non-terminaux de MARTHA en OCAML

```
type terminal =
```

```
| Int  
| Identifier  
| Lparen | Rparen | Comma  
| Plus | Minus | Star | Slash  
| Sum | From | To | Do | Def
```

```
type nonterminal =
```

```
| Expression  
| Term  
| Factor  
| SExpressions  
| SIdentifiers  
| Definition  
| Definitions  
| Program
```

Grammaire de MARTHA en OCAML

```
let martha = [  
  Expression -->  
  ! [ NonTerminal Expression ; Terminal Plus ; NonTerminal Term ]
```


Produire un mot à partir d'une grammaire

- En suivant les règles de la grammaire de MARTHA, on peut engendrer un programme syntaxiquement correct de ce langage :

Forme intermédiaire	Règle utilisée
<i>Program</i>	
<i>Definitions Expression</i>	$Program \rightarrow Definitions Expression$
<i>Expression</i>	$Definitions \rightarrow \epsilon$
<i>Expression '+' Term</i>	$Expression \rightarrow Expression '+' Term$
<i>Expression '+' Factor</i>	$Term \rightarrow Factor$
<i>Expression '+' 21</i>	$Factor \rightarrow i$
<i>Term '+' 21</i>	$Expression \rightarrow Term$
<i>Factor '+' 21</i>	$Term \rightarrow Factor$
<i>21 '+' 21</i>	$Factor \rightarrow i$

Produire un mot à partir d'une grammaire

- On aurait pu obtenir le même programme par un chemin différent :

Forme intermédiaire	Règle utilisée
<i>Program</i>	
<i>Definitions Expression</i>	$Program \rightarrow Definitions Expression$
<i>Expression</i>	$Definitions \rightarrow \epsilon$
<i>Expression '+' Term</i>	$Expression \rightarrow Expression '+' Term$
<i>Term '+' Term</i>	$Expression \rightarrow Term$
<i>Term '+' Factor</i>	$Term \rightarrow Factor$ (à l'occurrence 2)
<i>Term '+' 21</i>	$Factor \rightarrow i$
<i>Factor '+' 21</i>	$Term \rightarrow Factor$
<i>21 '+' 21</i>	$Factor \rightarrow i$

(Notons que l'occurrence précise d'application d'une règle peut être nécessaire.)

Les règles en toute généralité

- | Rien ne nous empêche d'écrire une règle « ADDAND » de la forme :
$$\textit{Identifier} \text{ ' , ' } \textit{Identifier} \text{ ') ' } \rightarrow \textit{Identifier} \textbf{ and } \textit{Identifier} \text{ ') ' }$$
- ⇒ En utilisant plusieurs symboles à la gauche d'une règle, on peut définir un **contexte** d'application de cette règle.

Dérivation

- | Soit une grammaire $G \equiv T, N, R, S$
- | Soient u et v , deux séquences de $T \cup N^*$.
- | On écrit « $u \rightarrow v$ », qui se lit « u dérive v », si :
 - ▶ la séquence u s'écrit « $w_1 i_1 \dots i_m w_2$ » ;
 - ▶ la séquence v s'écrit « $w_1 p_1 \dots p_n w_2$ » ;
 - ▶ la règle « $i_1 \dots i_m \rightarrow p_1 \dots p_n$ » est dans R .
- | En d'autres termes, v est u dont une sous-séquence a été réécrite par une règle de R .
- | La relation « \rightarrow » est appelée **relation de dérivation immédiate** de G .
- | La fermeture transitive et réflexive de cette relation est appelée **relation de dérivation** de G . Elle est notée « \rightarrow^* ».
- | Le langage engendré par G , que nous noterons généralement \mathcal{L}_G , est :

$$\{v \in T^* \mid S \rightarrow^* v\}$$

Dérivation en OCAML

(* Une application de règle est déterminée par *)
(* son lieu d'application et la règle appliquée. *)

type ('t, 'n) *rule_{application}* =
 int × ('t, 'n) rule

(* Une dérivation peut être représentée par *)
(* une liste d'applications de règle. *)

type ('t, 'n) derivation =
 ('t, 'n) *rule_{application}* list

Dérivation en OCAML

```
let rulenumber g index =  
  List.nth g index  
  
let rulesabout (g ('t', 'n') grammar) t =  
  List.filter (function (  
    | (NonTerminal rt, []) , _ ) → rt = t  
    | _ → false)  
    g  
  
let ruleabout g?(index=0) t =  
  List.nth (rulesabout g t) index
```

Quelques accesseurs de règle dans une grammaire.

Dérivation en OCAML

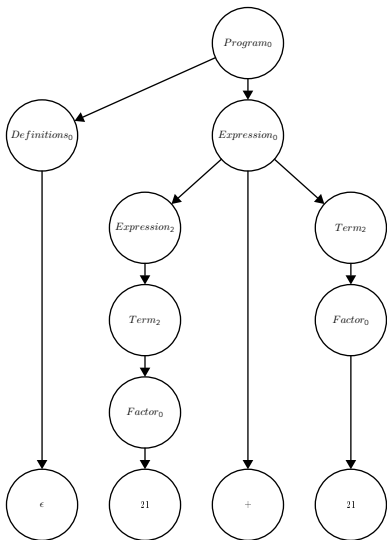
```
let derivationexample = [  
  0, ruleabout martha Program ;  
  0, ruleabout martha ~index 0 Definitions ;  
  0, ruleabout martha ~index 0 Expression ;  
  2, ruleabout martha ~index 2 Term ;  
  2, ruleabout martha ~index 0 Factor ;  
  0, ruleabout martha ~index 2 Expression ;  
  0, ruleabout martha ~index 2 Term ;  
  0, ruleabout martha ~index 0 Factor ;  
]
```

La dérivation, représentée à l'aide d'une valeur OCAML.

Dérivation en OCAML

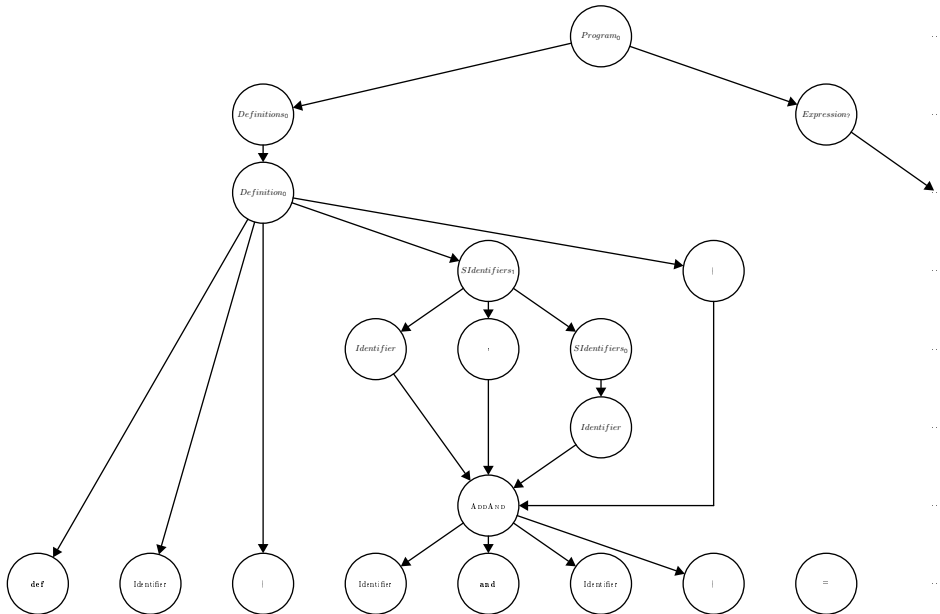
```
let apply_rule occ ((start, seq), result) s =  
  let prefix, focus = split occ s in  
  let suffix =  
    let rec aux (pattern, s) =  
      match (pattern, s) with  
      | [], suffix → suffix  
      | p, ps, sym, syms when p = sym → aux (ps, syms)  
      | _ → failwith "Invalid rule application."  
    in  
    aux (start, seq, focus)  
  in  
  prefix @ result @ suffix  
  
let rec interpret g s = function  
  | [] → s  
  | (occ, r) → interpret g (apply_rule occ r s) d
```


Représentation graphique



- | Un **graphe (acyclique) de production** associé à une dérivation est formé :
 - ▶ de nœuds correspondants aux règles appliquées ;
 - ▶ d'arêtes symbolisant les entrées et sorties des règles.
- | Par abus de notation, un nœud contenant un symbole correspond à la règle de reconnaissance de ce symbole.
- | Dans cette représentation, les deux dérivations précédentes sont identiques.

Exemple : une règle à plusieurs entrées



Construire la représentation graphique d'une dérivation

Exercice

Instrumentez l'interprétation des dérivations que nous avons écrit en OCAML de façon à construire un graphe de production.

(Vous pouvez produire un fichier dans le format d'entrée de GRAPHVIZ⁴.)

4. <http://www.graphviz.org/>

Représentations canoniques des graphes de production

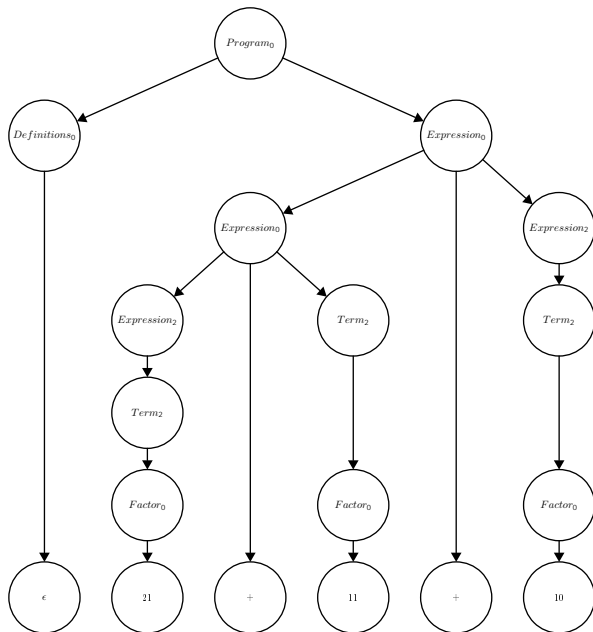
- | En effectuant un parcours *préfixe* du graphe de production, on obtient la **dérivation gauche**, c'est-à-dire la dérivation qui réécrit en priorité les symboles les plus à gauche.
 - | En effectuant un parcours *postfixe*, on obtient l'image miroir de la **dérivation droite**, c'est-à-dire la dérivation qui réécrit en priorité les symboles les plus à droite.
- ⇒ Ces dérivations sont des représentants canoniques pour les dérivations équivalentes (*i.e.* qui possèdent le même arbre de production).

La grammaire du langage MARTHA, modifiée

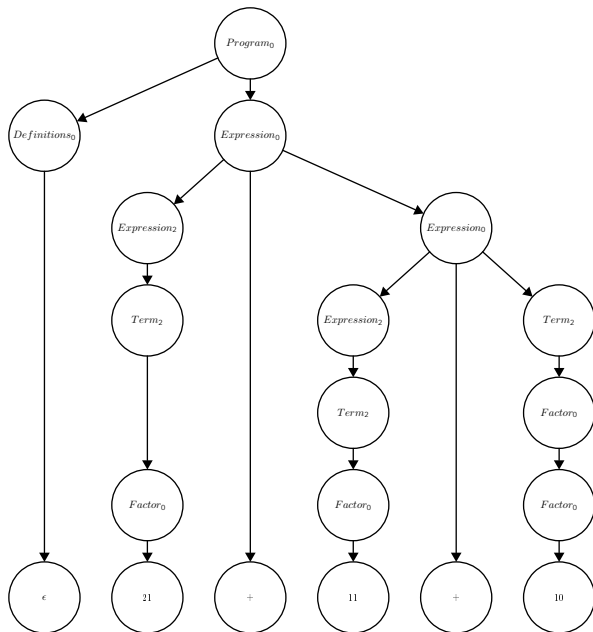
- On modifie la grammaire de MARTHA de la façon suivante :

<i>Expression</i>	→	<i>Expression</i> '+' <i>Expression</i>
<i>Expression</i>	→	<i>Expression</i> '-' <i>Expression</i>
<i>Expression</i>	→	<i>Term</i>
	...	

Exemple : une réelle ambiguïté



Exemple : une réelle ambiguïté



Ambiguïté

- | Une grammaire G est **ambiguë** si un mot est dérivable par deux arbres de production différents.
- | Dans l'exemple précédent, on a reconnu deux parenthésages différents de l'expression « $21 + 11 + 10$ » : « $(21 + 11) + 10$ » et « $21 + (11 + 10)$ ».
- | Ce type d'ambiguïté est bénigne dans le sens où les deux façons d'analyser l'expression conduisent à une même interprétation.
- | Cependant, certaines ambiguïtés ne le sont pas : les expressions « $21 - 11 - 10$ » et « $(21 - 11) - 10$ » valent 0 tandis que « $21 - (11 - 10)$ » vaut 20.

Spécification du problème de la reconnaissance

- | Le problème de l'analyse syntaxique se spécifie généralement ainsi :
 - ▶ Entrées :
Une grammaire $G \equiv (T, N, R, S)$, une entrée $I \in (T \cup N)^*$.
 - ▶ Précondition :
La grammaire G est non ambiguë.
 - ▶ Sortie :
Un arbre de production t ou bien une erreur.
 - ▶ Postcondition :
Si la sortie est un arbre t alors il correspond à la dérivation⁵ de I par les règles de G .
Sinon, il n'existe pas de dérivation de I par les règles de G .

5. Plus précisément à un ensemble de dérivations équivalentes

Spécification du problème de la reconnaissance

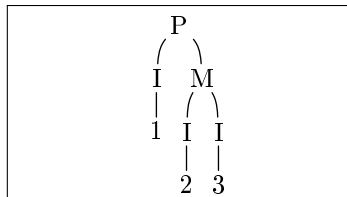
- | Le problème de l'analyse syntaxique peut aussi se spécifier ainsi :
 - ▶ Entrées :
Une grammaire $G \equiv (T, N, R, S)$, des règles de suppression des ambiguïtés, une entrée $I \in (T \cup N)^*$.
 - ▶ Précondition :
Pas de précondition.
 - ▶ Sortie :
Un arbre de production F ou bien une erreur.
 - ▶ Postcondition :
Si la sortie est un arbre de production alors il existe plusieurs dérivations de I par les règles de G mais les règles de suppression des ambiguïtés ont permis d'effectuer une discrimination parmi ces dérivations non équivalentes.
Sinon, il n'existe pas de dérivation de I par les règles de G .

Exemple de règles de suppression des ambiguïtés

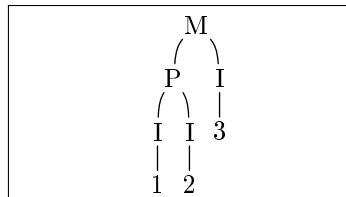
- Soit la grammaire suivante :

E	\rightarrow	i	(I)
		$E \text{ } ^{*} \text{ } E$	(M)
		$E \text{ } ^{+} \text{ } E$	(P)

- Voici deux arbres de syntaxe abstraite obtenus pour l'entrée « 1 + 2 * 3 » :



Dérivation « PIMII »



Dérivation « MPIII »

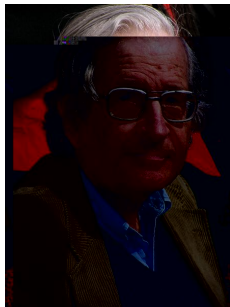
- La règle « la multiplication a une priorité plus forte que l'addition » s'exprime en termes de l'arbre de production :
« Un nœud “P” doit ne doit pas être le successeur direct d'un nœud “M”. »

Exemple de règle de suppression des ambiguïtés

Exercice

Comment rajouter les expressions parenthésées au langage précédent ? Est-ce que la règle de suppression des ambiguïtés fonctionne toujours ?

Hiérarchie de Chomsky



- | Dans les années 60, Noam Chomsky a défini une **classification** des grammaires génératives s'appuyant sur des restrictions progressives de la syntaxe des règles dans le but de faciliter leur utilisation sans pour autant restreindre de façon déraisonnable leur expressivité (leur pouvoir génératif).
- | Les grammaires de type 0 sont non restreintes.

Grammaire de type 1

- | Il y a deux définitions équivalentes des grammaires de type 1.
 - | Une grammaire est de type 1 (monotone) si elle ne contient aucune règle dont le membre gauche est plus long que le membre droit.
 - | Une grammaire est de type 1 (dépendante du contexte) si ses règles peuvent dépendre du contexte. Une règle est dépendante du contexte si elle est de la forme « $uSw \rightarrow uvw$ » où u , v et w sont des séquences de symboles (terminaux ou non terminaux) et S est un non terminal.
- ⇒ Trouver un exemple simple de langage qui ne peut pas être généré par une grammaire de type 1 est difficile.

Grammaire de type 2

- | Nous nous intéresserons surtout aux grammaires de type 2 car ce sont celles utilisées pour définir la syntaxe des langages de programmation.
- | Une grammaire est type 2 si toutes ses règles sont de la forme « $N \rightarrow w$ » où w est une séquence de symboles (terminaux et non terminaux) et N est un non terminal.
- | C'est une grammaire de type 1 dont toutes les règles ont des contextes vides.
- | On les appelle des grammaires **hors-contexte**.
- | Dans le graphe de production, ce que produit un nœud est indépendant de ce que produisent ses voisins : le graphe de production est donc un arbre.

Grammaire de type 2

- | On peut partitionner les règles d'une grammaire de type 2 suivant le non-terminal à gauche de la flèche. Ce sont les règles qui définissent ce non-terminal.
- | Soit un non-terminal A . Les règles de A peuvent être regroupées en une règle de la forme :

$$A \rightarrow w_1 \mid \dots \mid w_n$$

où chaque w_i est de la forme $v_1 \dots v_{k_i}$.

- | Dès lors, on peut définir le langage associé à ce non-terminal A :

$$\mathcal{L}_A = \mathcal{L}_{w_1} \cup \dots \cup \mathcal{L}_{w_n}$$

où $\mathcal{L}_{w_1} = \mathcal{L}_{v_1} \dots \mathcal{L}_{v_{k_i}}$, c'est-à-dire la concaténation des langages des symboles $v_i \ i \in 1 \dots k_i$.

- | Le langage \mathcal{L}_T associé à un terminal T est bien sûr « $\{T\}$ ».

Définition récursive

- | La restriction des grammaires hors-contexte n'interdit pas l'utilisation récursive d'un non-terminal A dans sa définition.
- | Ainsi, on peut décider si :
 - ▶ un non-terminal A est **récurif à gauche**, c'est-à-dire si il peut produire une chaîne commençant par le non-terminal A ;
 - ▶ un non-terminal A est **récurif à droite**, c'est-à-dire si il peut produire une chaîne se concluant par le non-terminal A ;
 - ▶ un non-terminal A est **auto-imbriqué**, c'est-à-dire si il peut produire une chaîne dans laquelle le terminal A se situe à gauche et à droite de deux séquences non vides.
- | Les grammaires hors-contextes permettent donc d'exprimer l'**imbrication**, une construction très courante des syntaxes de langage de programmation.

Notation BNF pour les grammaires hors-contextes

- | La notation BNF (Backus-Naur Form) est utilisée traditionnellement pour représenter les grammaires hors-contextes. Il en existe de très nombreuses variantes.
 - | Les non-terminaux sont écrits entre chevrons : « $\langle T \rangle$ ».
 - | Les flèches sont remplacées par le symbole « \rightarrow ».
 - | On regroupe les règles comme indiqué plus tôt.
 - | Il existe une version étendue de BNF (notée souvent EBNF) qui inclut les opérateurs “*” (zéro ou plusieurs fois), “+” (au moins une fois) et “?” (au plus une fois), ainsi que la possibilité de parenthéser les sous-séquences auxquelles on désire appliquer ces opérateurs.
- ⇒ Toutes ces notations sont équivalentes.

Spécification YACC de grammaire hors-contexte

- | Dans les années 70, de nombreuses recherches ont porté sur le problème de reconnaissance des langages décrits par les grammaires hors-contexte pour les appliquer à la construction des compilateurs.
- | L'outil YACC (*Yet Another Compiler Compiler*) a été développé à l'époque en s'appuyant sur l'algorithme d'analyse syntaxique appelé LALR(1), que nous aborderons dans ce cours.
- | Cet algorithme ne traite pas la totalité des grammaires hors-contextes mais la classe des langages LALR(1) est un bon compromis entre expressivité et efficacité de l'algorithme d'analyse.
- | Nous utiliserons une version plus moderne de YACC appelée MENHIR.
- | MENHIR est dédié au langage OCAML et reconnaît une classe plus large de langages, les langages LR(1).

Spécification YACC de grammaire hors-contexte

YACC a la spécification suivante :

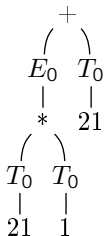
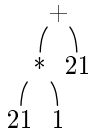
- ▶ Entrées :
Une grammaire hors-contexte G , un ensemble de règle de suppression d'ambiguïté.
- ▶ Précondition :
Pas de précondition.
- ▶ Sortie :
Un **programme** qui implémente un analyseur syntaxique pour G ou bien une liste de **conflits**.
- ▶ Postcondition :
Si la sortie est un programme alors cela signifie que la grammaire est dans la classe des langages LALR(1) et que les règles de suppression des ambiguïtés sont complètes. Sinon, cela signifie que les règles de suppression des ambiguïtés ne sont pas suffisantes ou que la grammaire n'est pas dans la classe LALR(1).

Transformations de grammaire

- | Plusieurs grammaires peuvent reconnaître le même langage.
 - | Il est parfois nécessaire de **transformer** une grammaire pour la rendre compatible avec la classe de grammaires traitée par un algorithme.
 - | Ces transformations nécessitent souvent l'ajout de nouveaux non-terminaux et la décomposition de certaines règles.
- ⇒ On s'éloigne parfois énormément de la grammaire initiale, la plus naturelle.
- | La syntaxe d'un langage de programmation est ainsi souvent décrite par une **grammaire hors-contexte de spécification** qui définit les différentes catégories syntaxiques du langage. Cette grammaire est à destination du programmeur puisqu'elle lui donne une vue synthétique des programmes syntaxiquement corrects. Elle sert aussi à définir la forme des arbres de syntaxe abstraite sur lesquels s'appuient les phases suivantes du compilateur.
 - | L'analyse syntaxique utilise une **grammaire d'analyse** compatible avec les algorithmes d'analyse syntaxique connus et reconnaissant le même langage que la grammaire de spécification.

De l'arbre de production à l'arbre de syntaxe abstraite

- La remarque précédente implique que l'arbre de production issu de l'analyse syntaxique fait référence aux règles de la grammaire d'analyse et est donc différent de l'arbre de syntaxe abstraite qui fait référence à la grammaire de spécification.
- Exemple :

$$\begin{array}{lcl} E & ::= & T \\ & | & E '+' T \\ T & ::= & F \\ & | & T '*' F \\ F & ::= & i \end{array}$$

$$\begin{array}{lcl} E & ::= & i \\ & | & E '*' E \\ & | & E '+' E \end{array}$$


Action sémantique

- | En pratique, peu de systèmes produisent *explicitement* l'arbre de production.
- | Les outils de la famille YACC exigent une **action sémantique** associée à chaque règle. Elle sert à calculer une **valeur sémantique** associée à chaque non-terminal reconnu en utilisant les valeurs sémantiques des non-terminaux intervenant de la règle qui le produit.
- | En MENHIR, on écrit ainsi :

```
expr  lhs=expr PLUS rhs=term
{ (* L'action sémantique est une expression OCaml. *)
  (* Elle peut faire référence à [lhs] et [rhs], *)
  (* Les valeurs sémantiques des deux sous-expressions. *)
  (* Ici, on utilise le constructeur [Add] du type [ast]. *)
  Add (lhs, rhs)
}
```

Format des spécifications écrites à l'aide de MENHIR

```
%{  
  (* Ici le prélude, c'est-à-dire du code OCaml qui peut définir *)  
  (* des types de données et des fonctions utilisés dans les *)  
  (* actions sémantiques. *)  
%}  
  
(* Déclarations de la grammaire. *)  
%token T (* Définit un token T. *)  
%token<int> I (* Définit un token I auquel est attaché un entier. *)  
  
%start e (* Déclare le non-terminal "e" d'entrée de la grammaire. *)  
          (* Menhir accepte plusieurs points d'entrée. *)  
%type<int> e (* Déclare le type de la valeur sémantique de e. *)  
  
(* Des priorités entre des règles (explications plus tard) *)  
%left p  
%right u v  
%nonassoc k  
%%  
  
(* Les règles de la grammaire. *)  
e : T x=e y=I { x + y }  
  | T T x=e y=I { x * y }  
  
%{  
  (* Postlude *)  
%}
```


Exemple de spécification écrite à l'aide de MENHIR

```
%{
  (* Abstract Syntax Tree. *)
  type exp =
    | Int of int
    | Add of exp × exp
    | Mult of exp × exp
}%

(* The lexing phase will produce the following tokens: *)
%token<int> INT
%token PLUS STAR EOF

(* Here are the declarations of non terminal symbols: *)
%type<exp> top_exp
%start top_exp

%%

(* Now, we are defining the rules of the grammar: *)
top_exp: e=exp EOF { e }

exp: x=INT { Int x }
    | lhs=exp PLUS rhs=exp { Add (lhs, rhs) }
    | lhs=exp STAR rhs=exp { Mult (lhs, rhs) }
```

Utilisation de MENHIR

- | On utilise la commande :

```
% menhir --explain arith.mly  
Warning 2 states have shift/reduce conflicts.  
Warning 4 shift/reduce conflicts were arbitrarily resolved.
```

⇒ MENHIR nous informe qu'il a bien réussi à interpréter notre grammaire mais en faisant des choix arbitraires pour résoudre certaines ambiguïtés.

- | MENHIR a produit :

- ▶ un fichier `arith.ml` qui réalise l'analyse syntaxique ;
- ▶ un fichier `arith.mli` qui définit son interface ;
- ▶ un fichier `arith.conflicts` qui explique comment la résolution des conflits a été faite.

⇒ Il y a très peu de chances pour que ce soit ces choix arbitraires soient les bons. Regardons ce dernier fichier !

Explication d'un conflit particulier par MENHIR

- ** Conflict (shift/reduce) in state 7.
- ** Tokens involved : STAR PLUS
- ** The following explanations concentrate on token STAR.
- ** This state is reached from top_exp after reading :

exp PLUS exp

- ** The derivations that appear below have the following common factor :
- ** (The question mark symbol (?) represents the spot where the derivations begin to differ.)

top_exp
exp EOF
(?)

- ** In state 7, looking ahead at STAR, reducing production
- ** $\text{exp} \rightarrow \text{exp PLUS exp}$
- ** is permitted because of the following sub-derivation :

exp STAR exp // lookahead token appears
exp PLUS exp .

- ** In state 7, looking ahead at STAR, shifting is permitted
- ** because of the following sub-derivation :

exp PLUS exp
exp . STAR exp

Fonctionnement d'un analyseur produit par MENHIR

- | Un analyseur syntaxique produit par MENHIR lit le flot de lexèmes de gauche à droite.
- ⇒ Il fait partie de la famille des analyseurs **directionnels**.
 - | Il essaie de reconstruire l'arbre de production en partant des feuilles.
- ⇒ C'est une analyse **ascendante**.
 - | Pour cela, il maintient une pile contenant les sous-arbres de l'arbre de production déjà reconnus et prend **une** décision en fonction du sommet de cette pile et du lexème suivant.
- ⇒ L'analyseur est un **automate à pile déterministe** qui lit un lexème en avant.
 - | Il y a deux types de décision :
 - ▶ La décision "avance" (*shift*) : on pousse le lexème suivant sur la pile.
 - ▶ La décision "réduit" (*reduce*) : on dépile N sous-arbres de la pile pour en construire un autre que l'on pose au sommet de la pile.

Résolution d'un conflit par MENHIR

- | Le conflit précédent s'explique ainsi :
 - | Après avoir reconnu « exp PLUS exp » (trois éléments sur la pile), si le prochain lexème est « STAR » alors, MENHIR ne sait pas choisir entre :
 - ▶ L'action "avance" : on empile *STAR*.
 - ▶ L'action "réduit" : on construit une nouvelle expression correspondant au sous-arbre « exp + exp » que l'on met sur la pile.
 - | De toute évidence, la seconde action n'est pas acceptable.
 - | Un autre conflit apparaît après avoir lu « exp STAR exp » et si le prochain lexème est « PLUS ». Dans ce cas, on veut plutôt réduire.
- ⇒ La règle de réduction de "*" doit avoir une priorité plus forte que celle de " ". Il suffit d'écrire :

```
(* On utilise le terminal le plus à droite de la règle *)  
(* pour parler de celle-ci. Nous verrons un mot-clé %prec, plus élégant. *)  
%nonassoc PLUS (* Les règles du haut sont moins prioritaires ... *)  
%nonassoc STAR (* ... que les règles du bas. *)
```

- ⇒ 2 conflits sont traités sur 4.

Résolution des conflits par MENHIR

- | Les 2 autres conflits parlent de l'interaction entre PLUS/PLUS et STAR/STAR.

Exercice

Sur quoi porte ces conflits ? Quel effet a eu le mot-clé %nonassoc ?

Grammaire de type 3

- | Les grammaires hors-contexte décrivent des langages dans lesquels les mots peut-être **imbriqués**. Lorsque l'on analyse un sous-mot d'un mot de ces langages, la forme des règles de la grammaire nous permet de nous **souvenir** de ce qui doit suivre le sous-mot une fois qu'il sera reconnu.
- | La restriction des grammaires de type 3 supprime cette mémoire.
- | On formalise cette restriction de la façon suivante :

Une règle ne peut produire qu'un ou plusieurs terminaux suivis d'un unique non terminal optionnel.

⇒ En d'autres termes, les règles doivent être linéaires droites.

- | Ces grammaires correspondent aux **langages rationnels** (ou réguliers) que vous avez déjà étudiés.
- | Les automates finis sont les outils de prédilection pour la construction des analyseurs syntaxiques de ces langages.
- | L'expressivité des langages rationnels n'est en général pas suffisante pour décrire la syntaxe des langages de programmation. Par contre, elle suffit à l'analyse lexicale.

Grammaire de type 4

- | La restriction des grammaires de type 4 interdit la présence d'un non-terminal à droite de la flèche.
- ⇒ Ces grammaires représentent des langages finis.
- | Elles fournissent un cadre pour décrire des énumérations.

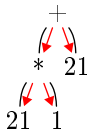
Deux directions possibles pour l'analyse syntaxique

┆ Rappel :

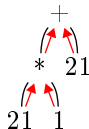
Le problème de l'analyseur syntaxique est de construire l'arbre de production d'un mot w par une grammaire non ambiguë G .

┆ Il y a deux grandes techniques pour résoudre ce problème :

- ▶ L'analyse **descendante** essaie d'imiter la dérivation (supposée) du mot w en partant du symbole d'entrée de la grammaire G en reconstruisant l'arbre de production **de sa racine vers ses feuilles**.
- ▶ L'analyse **ascendante** tente une reconstruction de la dérivation **à partir des feuilles** de l'arbre (le mot w) en **remontant vers la racine**.



Analyse descendante



Analyse ascendante

Exemple d'analyse descendante

- Soit la grammaire (tirée de *Parsing Techniques* de Grune et Jacobs) :

1	$S \rightarrow aSQ$
2	$S \rightarrow abc$
3	$bQc \rightarrow bbcc$
4	$cQ \rightarrow Qc$

et l'entrée : « *aabbcc* »

⑤

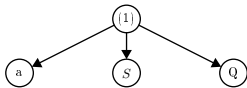
- L'analyse débute avec celle du symbole S_s .
- Quelle peut être la prochaine étape ?

Exemple d'analyse descendante

- Soit la grammaire (tirée de *Parsing Techniques* de Grune et Jacobs) :

1	$S \rightarrow aSQ$
2	$S \rightarrow abc$
3	$bQc \rightarrow bbcc$
4	$cQ \rightarrow Qc$

et l'entrée : « *aabbcc* »



- Comme « *abc* » n'est pas un préfixe de l'entrée, la règle (2) ne s'applique pas.
- Seule la règle (1) peut s'appliquer.

Exemple d'analyse descendante

- Soit la grammaire (tirée de *Parsing Techniques* de Grune et Jacobs) :

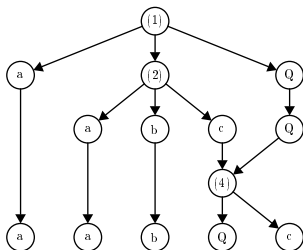
$$\begin{array}{l|l} 1 & S \rightarrow aSQ \\ 2 & S \rightarrow abc \\ 3 & \end{array}$$

Exemple d'analyse descendante

| Soit la grammaire (tirée de *Parsing Techniques* de Grune et Jacobs) :

1	$S \rightarrow aSQ$
2	$S \rightarrow abc$
3	$bQc \rightarrow bbcc$
4	$cQ \rightarrow Qc$

et l'entrée : « *aabbcc* »



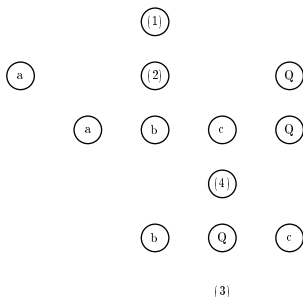
| Seule la règle (4) s'applique.
(C'était difficile à prévoir !)

Exemple d'analyse descendante

- Soit la grammaire (tirée de *Parsing Techniques* de Grune et Jacobs) :

1	$S \rightarrow aSQ$
2	$S \rightarrow abc$
3	$bQc \rightarrow bbcc$
4	$cQ \rightarrow Qc$

et l'entrée : « *aabbcc* »



Exemple d'analyse ascendante

- Soit la grammaire (tirée de *Parsing Techniques* de Grune et Jacobs) :

1	$S \rightarrow aSQ$
2	$S \rightarrow abc$
3	$bQc \rightarrow bbcc$
4	$cQ \rightarrow Qc$

et l'entrée : « *aabbcc* »

(a) (a) (b) (b) (c) (c)

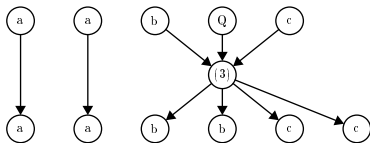
- Quelle règle peut produire une sous-séquence de non-terminaux de ce mot ?

Exemple d'analyse ascendante

- Soit la grammaire (tirée de *Parsing Techniques* de Grune et Jacobs) :

1	$S \rightarrow aSQ$
2	$S \rightarrow abc$
3	$bQc \rightarrow bbcc$
4	$cQ \rightarrow Qc$

et l'entrée : « *aabbcc* »



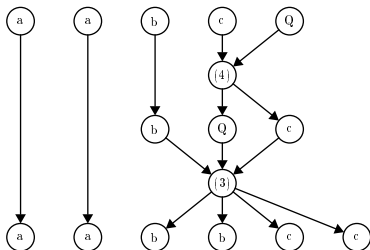
- De même, seule la règle (4) a pu produire le mot « *Q c* ».

Exemple d'analyse ascendante

Soit la grammaire (tirée de *Parsing Techniques* de Grune et Jacobs) :

1	$S \rightarrow aSQ$
2	$S \rightarrow abc$
3	$bQc \rightarrow bbcc$
4	$cQ \rightarrow Qc$

et l'entrée : « *aabbcc* »



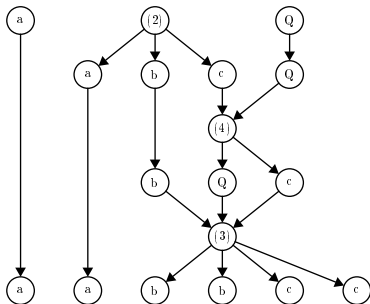
Ici, seule la règle (3) s'applique et elle mène à l'entrée.

Exemple d'analyse ascendante

Soit la grammaire (tirée de *Parsing Techniques* de Grune et Jacobs) :

1	$S \rightarrow aSQ$
2	$S \rightarrow abc$
3	$bQc \rightarrow bbcc$
4	$cQ \rightarrow Qc$

et l'entrée : « *aabbcc* »



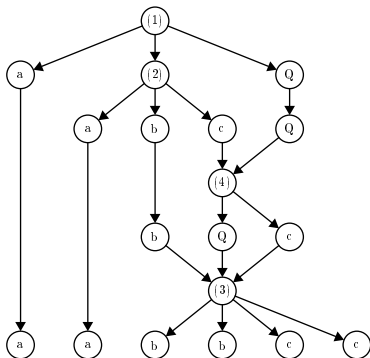
Encore une fois, une seule règle s'applique, la règle (2).

Exemple d'analyse ascendante

Soit la grammaire (tirée de *Parsing Techniques* de Grune et Jacobs) :

1	$S \rightarrow aSQ$
2	$S \rightarrow abc$
3	$bQc \rightarrow bbcc$
4	$cQ \rightarrow Qc$

et l'entrée : « *aabbcc* »



Pour finir, seule la règle (1) produit le mot « *aSQ* ».

Synthèse

Grammaire et analyse syntaxique

- | Nous avons (re)vu le concept de grammaire algébrique.
- | Le problème de l'analyse syntaxique consiste à construire un arbre de production à partir d'une grammaire et d'un mot qui retrace la reconnaissance de ce mot par la grammaire, c'est-à-dire sa dérivation.
- | MENHIR permet de spécifier des grammaires hors-contexte.
- | Il ne les traite pas dans leur totalité.
- | Il existe deux grands types d'analyse :

Les analyses descendantes et ascendantes

⇒ La définition de certains algorithmes qui les réalisent sera l'objet du prochain cours.

Bibliographie

- | *Parsing techniques, A practical guide*, Grune, Jacobs.
VU University Amsterdam, Amsterdam, The Netherlands
(Première édition est disponible en ligne : <http://www.few.vu.nl/~dick/PTAPG.html>)
- | *Théorie des langages*, Demaille, Yvon.
Notes de cours.
(<http://www.lrde.epita.fr/~akim/thl/theorie-des-langages-1.pdf>)

QCM : Question 1

L'analyse syntaxique :

- a. produit un flux de lexème.
- b. peut être ascendante ou descendante.
- c. s'appuie sur les grammaires de type 1.
- d. peut utiliser un automate (à pile).

QCM : Question 2

On peut résoudre un conflit dans une spécification de grammaire :

- a. en modifiant le langage à reconnaître.
- b. en réécrivant la grammaire en une grammaire équivalente.
- c. en renommant des non-terminaux.
- d. en spécifiant des priorités entre les règles.

QCM : Question 3

Un graphe de production est :

- a. l'arbre de syntaxe abstraite.
- b. un graphe potentiellement cyclique.
- c. un graphe acyclique.
- d. formé de nœuds qui dénotent des applications de règles de la grammaire.
- e. formé d'arêtes étiquetées par des noms de règles de la grammaire.

QCM : Question 4

L'analyse lexicale produit :

- a. un non-terminal.
- b. un flux de non-terminaux.
- c. un flux de terminaux.
- d. un flux de caractères.

QCM : Question 5

En parcourant l'arbre de production :

- a. suivant un parcours préfixe, on obtient une dérivation droite.
- b. suivant un parcours préfixe, on obtient une dérivation gauche.
- c. suivant un parcours infixe, on obtient une dérivation gauche.
- d. suivant un parcours infixe, on obtient une dérivation droite.

Introduction à la compilation

Cours 3 : Analyse syntaxique descendante

Yann Régis-Gianas
`yrg@pps.jussieu.fr`

PPS - Université Denis Diderot – Paris 7

Rappels sur l'analyse descendante

- Soit la grammaire (tirée de *Parsing Techniques* de Grune et Jacobs) :

(1)	\parallel	S	aSQ
(2)	\parallel	S	abc
(3)	\parallel	bQc	$bbcc$
(4)	\parallel	cQ	Qc

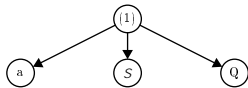
et l'entrée : « *aabbcc* »

Rappels sur l'analyse descendante

- Soit la grammaire (tirée de *Parsing Techniques* de Grune et Jacobs) :

(1)	S	aSQ
(2)	S	abc
(3)	bQc	$bbcc$
(4)	cQ	Qc

et l'entrée : « *aabbcc* »



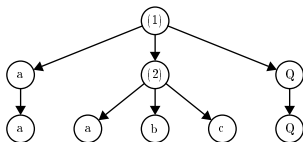
- Comme « *abc* » n'est pas un préfixe de l'entrée, la règle (2) ne s'applique pas.
- Seule la règle (1) peut s'appliquer.

Rappels sur l'analyse descendante

- Soit la grammaire (tirée de *Parsing Techniques* de Grune et Jacobs) :

(1)	S	aSQ
(2)	S	abc
(3)	bQc	$bbcc$
(4)	cQ	Qc

et l'entrée : « *aabbcc* »



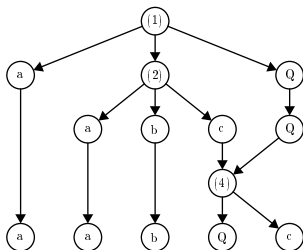
- Appliquer de nouveau la règle (1) imposerait le préfixe « *aaa* » dans l'entrée.
- On doit donc considérer la règle (2).

Rappels sur l'analyse descendante

- Soit la grammaire (tirée de *Parsing Techniques* de Grune et Jacobs) :

(1)	S	aSQ
(2)	S	abc
(3)	bQc	$bbcc$
(4)	cQ	Qc

et l'entrée : « *aabbcc* »



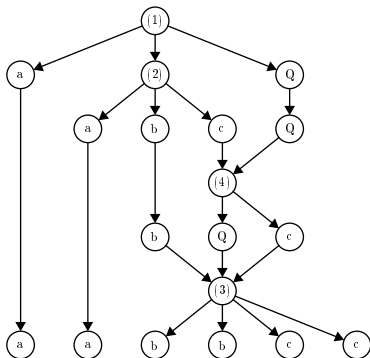
- Seule la règle (4) s'applique.
(C'était difficile à prévoir !)

Rappels sur l'analyse descendante

- Soit la grammaire (tirée de *Parsing Techniques* de Grune et Jacobs) :

(1)	S	aSQ
(2)	S	abc
(3)	bQc	$bbcc$
(4)	cQ	Qc

et l'entrée : « *aabbcc* »



- Ici, seule la règle (3) s'applique et elle mène à l'entrée.

Algorithmme de Unger

Principe

- ▶ Soit G , une grammaire **hors-contexte**, sans règle et sans boucle.
- ▶ Une façon naturelle (et naïve) de déterminer si une entrée $c_1 \dots c_n$ peut être produite par un non-terminal S consiste à :
 - ▶ **énumérer** l'ensemble des règles de G qui définissent S ;
 - ▶ Pour chaque règle de la forme « $S \rightarrow A_1 \dots A_m$ », pour **toutes les découpages** $\mathcal{P} \equiv P_1 \dots P_m$ de l'entrée en **m parties**, on détermine (par récursion) si A_i peut produire P_i .

Une application standard de la méthode « diviser pour régner ».

Exercice

1. Quelle est la complexité de cette méthode de recherche ?
2. Est-ce que cet algorithme termine ?

(réponses dans la suite)

Grammaire d'exemple

- On s'intéresse à la grammaire suivante :

$$\begin{array}{lcl} \textit{exp} & ::= & \textit{exp} + \textit{term} \\ & / & \textit{term} \\ \textit{term} & ::= & \textit{term} \textit{factor} \\ & / & \textit{factor} \\ \textit{factor} & ::= & 0 / 1 / \dots \\ & / & (\textit{exp}) \end{array}$$

Calculer toutes les partitions de N objets en M parties ?

Exercice

Comment calculer toutes les partitions de N objets en M parties (non vides) ? (en supposant $M > 0$)

Calculer toutes les partitions de N objets en M parties ?

Exercice

Comment calculer toutes les partitions de N objets en M parties (non vides) ? (en supposant $M > 0$)

```
let rec splitting m n l :  $\alpha$  splitting list =  
  assert (m > 0);  
  if (n < m) then []  
  else match m, l with  
  | 0, l →  
    assert false  
  | 1, l →  
    [[ l ]]  
  | m, l →  
    List.flatten (repeat 1 n  
      (fun k →  
        let (picked, l) = pick k l in  
        List.map (fun p → picked :: p)  
          (splitting (m - 1) (n - k) l)))
```

Implémentation (naïve) de Unger

```
let rec parse g s input : bool =  
  let rules : ('t, 'n) rule list = rulesabout g s in  
  let matchrhs ((_, (rhs : ('t, 'n) rhs))) =  
    let m = List.length rhs in  
    let ps = splitting m input in  
    let subinput_matchessymbol symbol subinput =  
      match symbol, subinput with  
      | Terminal t, [u] → t = u  
      | NonTerminal s, _ → parse g s subinput  
      | _ → false  
    in  
    let splitting_matchesrhs p =  
      List.forall2 subinput_matchessymbol rhs p  
    in  
    List.exists splitting_matchesrhs ps  
  in  
  List.exists matchrhs rules
```

Exercice

Quelle est la complexité de cette méthode de recherche ?

Règles de production vide

- Pour traiter les règles de production vide, on doit considérer les parties de taille 0 : peut-être suffit-il de modifier l'algorithme qui énumère les partitions ?

Exercice

Comment calculer toutes les partitions de N objets en M parties (possiblement vides) ?

```
let rec splitting m n l :  $\alpha$  splitting list =  
  match m, l with  
  | 0, []  $\rightarrow$  [ [] ]  
  | 0, _  $\rightarrow$  []  
  | 1, l  $\rightarrow$  [ [ l ] ]  
  | m, l  $\rightarrow$   
    List.flatten  
      (repeat 0 n  
        (fun k  $\rightarrow$   
          let (picked, l) = pick k l in  
          List.map (fun p  $\rightarrow$  picked :: p)  
            (splitting (m - 1) (n - k) l)))
```


Règles de production vide

- ▶ Par exemple, si on étend la grammaire à l'aide des règles :

<i>factor</i>	<i>INT plusplus</i>
<i>plusplus</i>	
<i>plusplus</i>	<i>BANG plusplus</i>

- ▶ L'algorithme boucle ! Pourquoi ?

Observation sur la trace d'exécution

- ▶ Le programme se pose sans cesse les mêmes questions.
- ▶ Si on cherche une dérivation pour « $1!$ » à partir de « exp » alors une des voies explorées par l'algorithme envisage la possibilité que cette entrée soit produite par la règle « $exp \rightarrow exp + term$ ». En étudiant la partition « $/1!$ », on doit résoudre le problème de la génération de « $1!$ » par « exp ». Ce dernier problème se réduit immédiatement sur lui-même, ce qui provoque la non terminaison de l'algorithme !

Forme des dérivations « qui bouclent »

- ▶ Les chemins de recherche de dérivation, à partir de S produisant w , qui sont exhibés par le phénomène précédent correspondent à des dérivations de la forme :

$$S \rightarrow \dots \rightarrow S$$

où \rightarrow et \rightarrow peuvent produire le mot vide.

- ▶ Si \rightarrow et \rightarrow ont effectivement produit le mot vide alors il existe une **infinité** de dérivations de cette forme : nous pouvons nous intéresser uniquement à celle **sans boucle**.
- ▶ Si \rightarrow et \rightarrow n'ont pas produit le mot vide alors il ne sert à rien de se poser la question « $S \stackrel{?}{\rightarrow} w$ » pour le S entre \rightarrow et \rightarrow .

On peut donc sans danger couper l'arbre de recherche en ce point.

Comment modifier l'algorithme pour prendre en compte cette remarque ?

Se donner une mémoire

- ▶ Une solution très simple – et dans cette situation, très efficace – consiste à se souvenir des réponses issues des analyses déjà effectuées.
- ▶ Quand on se pose la question :
« Est-ce que le non-terminal E peut produire w ? »
on vérifie d'abord dans une table si on ne s'est pas déjà posé cette question.
- ▶ Il y a alors trois cas possibles :

1. On ne s'est jamais posé la question :
Il faut faire le calcul et enregistrer le résultat dans la table.
2. On s'est déjà posé cette question :
On peut renvoyer le résultat enregistré dans la table.
3. On est déjà en train de se poser cette question :
Cela signifie qu'il y a une boucle : on peut s'arrêter et répondre que cette branche de recherche ne fournit pas de solution.

Se donner une mémoire en OCaml

```
type answer =  
  / BeingAnswered  
  / Answered of bool  
  
let h = Hashtbl.create 13 in  
  
let rec result_from_memory (s, input) =  
  try match Hashtbl.find h (s, input) with  
    / BeingAnswered    false  
    / Answered b       b  
  with Not_found  
    Hashtbl.add h (s, input) BeingAnswered ;  
    let y = traverse (s, input) in  
    Hashtbl.add h (s, input) (Answered y) ;  
    y
```

Algorithme de Unger avec mémoire

```
and traverse (s, input) : bool =  
  let rules = rulesabout g s in  
  let matchrhs ((_, (rhs : ('t, 'n) rhs)) as r) =  
    tracerule input r ;  
    let m = List.length rhs in  
    let ps = splitting m input in  
    let subinput_matchessymbol symbol input =  
      match symbol, input with  
      | Terminal t, [u] → t = u  
      | NonTerminal s, _ → result_frommemory (s, input)  
      | _ → false  
    in  
    let splitting_matchesrhs p =  
      tracesplitting p ;  
      List.for_all2 subinput_matchessymbol rhs p  
    in  
    List.exists splitting_matchesrhs ps  
  in  
  List.exists matchrhs rules  
in  
result_frommemory (s, input)
```

Procédé de « memoization »

- ▶ La méthode que nous venons d'employer s'apparente à de la **programmation dynamique** dans le sens où on réutilise les résultats de **sous-problèmes partagés** entre plusieurs problèmes pour factoriser les calculs.
- ▶ Cependant, il ne faut pas perdre de vue qu'on utilise un important espace mémoire.
- ▶ On peut écrire cette fonction en toute généralité de la façon suivante :

```
exception Loop

type  $\alpha$  answer = BeingAnswered | Answered of  $\alpha$ 

let rec memoize f =
  let h = Hashtbl.create 13 in
  fun x  $\rightarrow$ 
    try match Hashtbl.find h x with
      | BeingAnswered  $\rightarrow$  raise Loop
      | Answered b  $\rightarrow$  b
    with Not_found  $\rightarrow$ 
      Hashtbl.add h x BeingAnswered ;
      let y = f x in
      Hashtbl.add h x (Answered y) ;
      y
```


Inefficacité de l'algorithme de Unger

- ▶ Par une « simple » observation du premier caractère de l'entrée, on peut **prévoir** que l'étude de certaines règles ne va pas aboutir.
- ▶ Par exemple, en considérant *la taille minimale d'un mot produit par le côté droit d'une règle*, on déduit que les règles suivantes ne peuvent pas produire l'entrée associée :

$\text{expr} \rightarrow \text{expr} + \text{term} / (\text{INT})$
✓ $\text{expr} \rightarrow \text{expr} + \text{term} /$
✓ $\text{expr} \rightarrow \text{term} /$
✓ $\text{term} \rightarrow \text{term} \times \text{factor} /$
✓ $\text{term} \rightarrow \text{factor} /$
✓ $\text{factor} \rightarrow (\text{expr}) /$
✓ $\text{factor} \rightarrow \text{INT} /$
✓ $\text{expr} \rightarrow \text{expr} + \text{term} / ($
 $\text{expr} \rightarrow \text{term} / ($
✓ $\text{term} \rightarrow \text{term} \times \text{factor} / ($
 $\text{term} \rightarrow \text{factor} / ($
 $\text{factor} \rightarrow (\text{expr}) / ($
 $\text{factor} \rightarrow \text{INT} / ($
✓ $\text{expr} \rightarrow \text{expr} + \text{term} / (\text{INT}$
 $\text{expr} \rightarrow \text{term} / (\text{INT}$

✓ $\text{term} \rightarrow \text{term} \times \text{factor} / (\text{INT}$
 $\text{term} \rightarrow \text{factor} / (\text{INT}$
✓ $\text{factor} \rightarrow (\text{expr}) / (\text{INT}$
✓ $\text{expr} \rightarrow \text{expr} + \text{term} / \text{INT}$
 $\text{expr} \rightarrow \text{term} / \text{INT}$
✓ $\text{term} \rightarrow \text{term} \times \text{factor} / \text{INT}$
 $\text{term} \rightarrow \text{factor} / \text{INT}$
✓ $\text{factor} \rightarrow (\text{expr}) / \text{INT}$
 $\text{factor} \rightarrow \text{INT} / \text{INT}$
✓ $\text{factor} \rightarrow \text{INT} / (\text{INT}$
 $\text{expr} \rightarrow \text{term} / (\text{INT}$
 $\text{term} \rightarrow \text{term} \times \text{factor} / (\text{INT}$
 $\text{term} \rightarrow \text{factor} / (\text{INT}$
 $\text{factor} \rightarrow (\text{expr}) / (\text{INT}$

Analyse descendante prédictive

Analyse prédictive

- ▶ L'algorithme de Unger est **non directionnel**.
- ▶ Il se donne la possibilité de lire l'entrée plusieurs fois et d'une façon arbitraire. Il faut donc avoir sous la main la totalité de l'entrée.
- ▶ Nous allons nous restreindre à une lecture de l'entrée de **gauche à droite**.
- ▶ Dans ce cadre, on peut modéliser l'**analyse prédictive** ainsi :

Reste de l'entrée ...
Prédiction ...

- ▶ Le reste de l'entrée est formé de terminaux.
- ▶ La prédiction est une phrase intermédiaire contenant des symboles terminaux et non-terminaux.

Analyse prédictive : exemple

- Soit la grammaire :

$$\begin{aligned} S &::= aB \mid bA \\ A &::= a \mid aS \mid bAA \\ B &::= b \mid bS \mid aBB \end{aligned}$$

Exercice

Quel est le langage reconnu par cette grammaire ?

Analyse prédictive : exemple

$$\begin{aligned} S &::= aB \mid bA \\ A &::= a \mid aS \mid bAA \\ B &::= b \mid bS \mid aBB \end{aligned}$$

a a b b
S

- Essayons de calculer la dérivation de « aabb » par S .

Analyse prédictive : exemple

$$\begin{aligned} S &::= aB \mid bA \\ A &::= a \mid aS \mid bAA \\ B &::= b \mid bS \mid aBB \end{aligned}$$

a a b b
S

- ▶ On doit remplacer S par l'un de ses membres droits.
- ▶ Par observation du premier lexème "a", seule la première règle est applicable.

Analyse prédictive : exemple

$$\begin{aligned} S &::= aB \mid bA \\ A &::= a \mid aS \mid bAA \\ B &::= b \mid bS \mid aBB \end{aligned}$$

a	a b b
a B	

- ▶ La prédiction et l'entrée commence par le même symbole.
- ▶ On **accepte** le terminal "a".

Analyse prédictive : exemple

$$\begin{aligned} S &::= aB \mid bA \\ A &::= a \mid aS \mid bAA \\ B &::= b \mid bS \mid aBB \end{aligned}$$

a	a	b b
a	B	

- ▶ Le premier symbole de la prédiction est un non-terminal.
- ▶ Trois possibilités mais une seule est compatible avec le lexème "a".

Analyse prédictive : exemple

$$S ::= aB \mid bA$$
$$A ::= a \mid aS \mid bAA$$
$$B ::= b \mid bS \mid aBB$$

a	a	b b
a	a	B B

- On accepte de nouveau de nouveau le lexème “a”.

Analyse prédictive : exemple

$$\begin{aligned} S &::= aB \mid bA \\ A &::= a \mid aS \mid bAA \\ B &::= b \mid bS \mid aBB \end{aligned}$$

a a	b	b
a a	B	B

- ▶ Par observation du premier lexème de l'entrée, il reste deux choix.
- ▶ Cependant, si on choisit « bS », on aurions au moins un nouveau “a” à reconnaître. Or, la suite de l'entrée ne contient plus de “a”.

Analyse prédictive : exemple

$$\begin{aligned} S &::= aB \mid bA \\ A &::= a \mid aS \mid bAA \\ B &::= b \mid bS \mid aBB \end{aligned}$$

a a	b	b
a a	b	B

- On accepte le lexème “b”.

Analyse prédictive : exemple

$$S ::= aB \mid bA$$
$$A ::= a \mid aS \mid bAA$$
$$B ::= b \mid bS \mid aBB$$

a a b	b
a a b	B

- On doit encore appliquer première règle de B.

Analyse prédictive : exemple

$$\begin{aligned} S &::= aB \mid bA \\ A &::= a \mid aS \mid bAA \\ B &::= b \mid bS \mid aBB \end{aligned}$$

a a b b	
a a b b	

- ▶ On accepte finalement le dernier lexème “b”.
La prédiction est vide, tout comme l’entrée : l’analyse est un succès.
- ▶ La dérivation est :

$$S \quad aB \quad aaBB \quad aabB \quad aabb$$

Caractéristiques de l'analyse prédictive gauche-droite

- ▶ Nous avons produit la dérivation **gauche**.
- ▶ L'algorithme d'analyse est une succession de deux actions distinctes :
 - ▶ Soit la prédiction commence par un terminal "a" :
 - ▶ Si l'entrée commence aussi par "a" alors on continue.
 - ▶ Sinon on échoue.
 - ▶ Soit la prédiction commence par un non-terminal "A", alors on remplace ce non-terminal dans la prédiction par la partie droite de ses règles.
- ▶ Dans notre exemple, nous avons su choisir "magiquement" une seule partie droite à chaque étape. Cependant, en toute généralité, cet algorithme est encore un **processus de recherche** qui doit envisager toutes les possibilités.

Un modèle de calcul : l'automate à pile

- ▶ Les deux actions de l'algorithme d'analyse prédictif s'apparentent aux deux mécanismes sur lesquels s'appuie les **automates à pile**.
- ▶ Un automate à pile est une machine qui, en fonction d'un symbole d'entrée et du sommet de sa pile, empile ou dépile des symboles sur cette dernière.
- ▶ Un automate à pile **accepte** une séquence de symboles si elle mène à la pile vide.
- ▶ Il peut y avoir plusieurs choix d'action à effectuer pour un même symbole d'entrée. Dans ce cas, on parle d'automate non déterministe.

Définition formelle des automates à pile utilisés ici

- ▶ Nous allons utiliser la formulation suivante de la définition des automates à pile.
- ▶ Un automate qui suit un alphabet d'entrée (les terminaux) et un alphabet de pile (les non-terminaux et les terminaux) est la donnée de règles de la forme :

$$(i, s) \quad s_1 \dots s_n$$

où i est soit un terminal, soit vide.

s est un terminal ou un non-terminal.

s_i est un terminal ou un non-terminal pour tout i .

Exemple d'automates à piles

- Le langage de la grammaire :

$$S ::= aB \mid bA$$

$$A ::= a \mid aS \mid bAA$$

$$B ::= b \mid bS \mid aBB$$

- est reconnu par l'automate à pile **non déterministe** :

$$(\,, S) \qquad aB$$

$$(\,, S) \qquad bA$$

$$(\,, A) \qquad a$$

$$(\,, A) \qquad aS$$

$$(\,, A) \qquad bAA$$

$$(\,, B) \qquad b$$

$$(\,, B) \qquad bS$$

$$(\,, B) \qquad aBB$$

$$(a, a)$$

$$(b, b)$$

Environnement d'évaluation de l'automate à pile

- ▶ Un automate à pile ne nécessite que l'état de sa pile et son entrée pour fonctionner.
- ▶ Toutefois, pour mieux comprendre les algorithmes d'analyse syntaxique les utilisant, nous allons nous donner un environnement d'évaluation plus riche, qui se souvient des analyses déjà effectuées par l'automate.
- ▶ Pour cela, on définit une **description instantanée** par un tableau :

Entrée déjà analysée	Entrée restante
Règles utilisées	Prédictions

- ▶ La seconde ligne de ce tableau peut contenir plusieurs couples (analyse, prédiction) concurrents.
- ▶ On introduit un marqueur final '#' à la fin de l'entrée et de la prédiction.
- ▶ Ainsi, une description instantanée qui accepte l'entrée correspond une acceptation de '#'.

Pas de cas particulier pour vérifier le succès à la fin de l'analyse.

Domaine de recherche des dérivations

- ▶ Le domaine de recherche des dérivations peut donc se modéliser par un arbre dont les nœuds sont les descriptions instantanées et chaque arête correspond à un choix d'expansion d'une certaine règle d'un non-terminal.
- ▶ On choisit d'en faire un parcours **en profondeur** ou un parcours **en largeur**.

Exercice

Est-ce que cet arbre est **fini** ? À **branchement fini** ?

Un cas de non-terminaison

- Considérons la grammaire :

$$S ::= Sb / a$$

- Sur l'entrée « ab », on s'engage dans une **séquence infinie de prédictions** :

	<i>ab#</i>
	<i>S#</i>

	<i>ab#</i>
<i>S₁</i>	<i>Sb#</i>
<i>S₂</i>	<i>ab#</i>

	<i>ab#</i>
<i>S₁S₁</i>	<i>Sbbb#</i>
<i>S₁S₂</i>	<i>ab#</i>
<i>S₂</i>	<i>ab#</i>

	<i>ab#</i>
<i>S₁S₁S₁</i>	<i>Sbbbb#</i>
<i>S₁S₁S₂</i>	<i>abb#</i>
<i>S₁S₂</i>	<i>ab#</i>
<i>S₂</i>	<i>ab#</i>

...

Problème de la récursion à gauche

- ▶ Un non-terminal qui peut dériver une phrase commençant par lui-même est **récurif gauche**.
- ▶ Il y a deux sortes de récursion gauche, la **récursion gauche immédiate (ou directe)** d'un non-terminal A apparaît lorsqu'une règle de A possède un membre droit débutant par A .
- ▶ Un non-terminal récurif gauche qui n'est pas récurif gauche de façon immédiate, l'est de façon **indirect**.

Bonne nouvelle : on peut toujours **supprimer la récursion gauche**, c'est-à-dire transformer une grammaire possédant des non-terminaux récurifs à gauche en une grammaire équivalente non réursive gauche.

Élimination de la récursion gauche

Élimination de la récursion directe

- ▶ Nous allons voir une méthode de suppression de la récursion immédiate à gauche qui suppose que la grammaire considérée ne contient pas de règles de la forme :
 - ▶ « $A \rightarrow \epsilon$ » : règle de production vide.
 - ▶ « $A \rightarrow B$ » : règle unitaire.
où les symboles A et B sont des non-terminaux.

Heureusement, il est aussi possible de transformer toute grammaire avec de telles règles en une grammaire équivalente sans règles unitaires ou de production vide.

Élimination des règles de production vide

- Soit une grammaire G contenant une règle de la forme :

$$A$$

- Pour toute règle de la forme :

$$B \rightarrow A$$

- On peut rajouter la règle suivante dans la grammaire :

$$B$$

- On introduit alors un non-terminal A' avec les mêmes règles que A sauf celles de production vide. On peut remplacer toutes les occurrences de A par A' dans les règles de la grammaire.

Terminaison ?

- ▶ Le processus suivant peut introduire de nouvelles règles de production vide.
- ▶ On doit donc **itérer** le processus pour supprimer toutes ces règles.

Exercice

Est-ce que cet algorithme termine ?

Terminaison de la suppression des règles de production vide

- ▶ Le nombre de non-terminaux de la grammaire qui produisent ϵ est fini.
- ▶ Ce nombre décroît à chaque itération.

Élimination des règles unitaires

- Soit une grammaire G possédant une règle :

$$A \rightarrow B$$

où A et B sont des non-terminaux.

- Supposons que B soit défini ainsi :

$$B \rightarrow \epsilon \mid x_1 \mid x_2 \mid \dots \mid x_n$$

- On peut mettre en ligne la définition de B en rajoutant la règle :

$$A \rightarrow \epsilon \mid x_1 \mid x_2 \mid \dots \mid x_n$$

Terminaison ?

- ▶ Encore une fois, cette transformation de grammaires peut introduire de nouvelles règles unitaires.

Exercice

Est-ce que cet algorithme termine ?

Terminaison de l'élimination des règles unitaires

- ▶ On peut retomber sur la règle « $A \rightarrow B$ ».
- ▶ Ce cas témoigne de la présence de dérivations infinies.
- ▶ On peut, sans modifier le langage, supprimer cette nouvelle occurrence de la règle « $A \rightarrow B$ ».

Élimination des règles unitaires et de production vide :

Exemple

```
exp ::= exp + term
      / term
term ::= term factor
      / factor
factor ::= int plusplus
        / ( exp )
plusplus ::=
        / ! plusplus
```

Élimination des règles unitaires et de production vide :

Exemple

```
exp ::= exp + term
      / term factor
      / i
      / i plusplus'
      / ( exp )
term ::= term factor
      / i
      / i plusplus'
      / ( exp )
factor ::= i
        / i plusplus'
        / ( exp )
plusplus' ::= ! plusplus'
```

Élimination de la récursion immédiate gauche

- Soit une grammaire G , sans règle unitaire, ni règle de production vide.
- Pour tout non-terminal (récursif immédiat), on peut partitionner ses règles ainsi :

$$\begin{array}{l} A \\ A \end{array} \quad \begin{array}{l} A_1 / \dots / A_m \\ A_1 / \dots / A_n \end{array}$$

où $A_i \in (T \cup N)^+$.

$A_i \in (T \cup N)^+$ et ne commencent pas par le non-terminal A .

- On peut remplacer cet ensemble de règles par les règles suivantes :

$$\begin{array}{l} A \\ A_{\text{tête}} \\ A_{\text{suite}} \\ A_{\text{suites}} \end{array} \quad \begin{array}{l} A_{\text{tête}} A_{\text{suites}} / A_{\text{tête}} \\ A_1 / \dots / A_n \\ A_1 / \dots / A_m \\ A_{\text{suite}} A_{\text{suites}} / A_{\text{suite}} \end{array}$$

où $A_{\text{tête}}$, A_{suite} et A_{suites} sont des non-terminaux fraîchement introduits.

Élimination de la récursion immédiate : exemple

$$\begin{aligned} \text{exp} &::= \text{exp} + \text{term} \\ &/ \quad \text{term} \quad \text{factor} \\ &/ \quad \mathbf{i} \\ &/ \quad \mathbf{i} \text{ plusplus}' \\ &/ \quad (\text{exp}) \end{aligned}$$
$$\begin{aligned} \text{exp}_{\text{tête}} &::= \text{term} \quad \text{factor} \\ &/ \quad \mathbf{i} \\ &/ \quad \mathbf{i} \text{ plusplus}' \\ &/ \quad (\text{exp}) \\ \text{exp}_{\text{suite}} &::= + \text{term} \\ \text{exp}_{\text{suites}} &::= \text{exp}_{\text{suite}} \text{exp}_{\text{suites}} \\ \text{exp} &::= \text{exp}_{\text{tête}} \text{exp}_{\text{suites}} / \text{exp}_{\text{tête}} \end{aligned}$$

Élimination de la récursion gauche

- ▶ Dans notre cas, il suffit de supprimer la récursion gauche immédiate de *exp* et *term* pour supprimer toutes récursions gauches de notre grammaire.
- ▶ Pour supprimer *toute récursion gauche indirecte*, il faut itérer ce procédé en suivant l'ordre induit par la relation de dépendance entre non-terminaux définie de la façon suivante :

A dépend en tête de B si il existe une règle « $B \rightarrow A \dots$ ».

Checkpoint

- ▶ À ce stade, nous savons donc construire un automate à pile **non déterministe** pour toute grammaire hors-contexte (éventuellement préalablement transformée en une grammaire équivalente adéquate).
- ▶ On peut implanter cet automate sous la forme d'un programme composée de fonctions mutuellement récursives correspondant à l'analyse de chaque non-terminal.
- ▶ La pile des appels récursifs modélise alors les points de choix de l'arbre de recherche.

La complexité en pire cas est exponentielle.

On aimerait transformer le processus de recherche en un algorithme déterministe, quitte à restreindre la classe des grammaires reconnues.

Analyse descendante prédictive dirigée par une table

Sources d'information pour diriger la recherche

- ▶ Pour transformer notre méthode de recherche en un algorithme déterministe, il faut trouver les sources d'information qui permettent de faire un choix (et le bon) sans avoir à revenir dessus.
- ▶ Il y a deux sources essentielles d'information pour décider qu'elle est la bonne règle de grammaire à utiliser :
 - ▶ La dérivation qui a été construite jusqu'à maintenant (la partie analysée) ;
 - ▶ Les premiers lexèmes de l'entrée restant à analyser.

Cas simpliste

- ▶ On peut construire la table suivante pour diriger l'analyse syntaxique :

	a	b	$\#$
S	$S_1 \quad aB$	$S_2 \quad bA$	
A	$A_1 \quad a$ $A_2 \quad aS$	$A_3 \quad bAA$	
B	$B_3 \quad aBB$	$B_1 \quad b$ $B_2 \quad bS$	

- ▶ Une fois cette **table d'analyse syntaxique** construite, la grammaire n'est plus nécessaire : les étapes de prédiction utilisent désormais une table.
- ▶ Si l'analyse pointe sur une case sans règle alors l'entrée est rejetée.
- ▶ Avoir plusieurs règles dans une case rend l'analyse non déterministe.

La classe des grammaires LL(1)

- ▶ Une grammaire est LL(1) si sa table d'analyse syntaxique contient au plus une règle par case.
- ▶ L'appellation « LL(1) » s'explique ainsi :
 - ▶ *Left to right* : une analyse directionnelle qui lit l'entrée de gauche à droite.
 - ▶ *Leftmost* : on imite la dérivation gauche.
 - ▶ (1) : on lit un lexème en avant pour prendre ses décisions.

La classe des grammaires SLL(1)

- ▶ La restriction « toute règle commence par un terminal » est trop restrictive.
- ▶ La classe des grammaires la respectant s'appelle *Simple LL(1)* (SLL(1)).
La fin de cette séance va viser à relâcher cette restriction.

Analyse descendante prédictive LL(1) totale

L'idée à retenir

- ▶ Les grammaires SLL(1) permettent de calculer aisément la table d'analyse syntaxique : il suffit de lire chaque règle et de la placer dans la bonne case de la table en observant le premier lexème de sa partie droite.
- ▶ On peut généraliser cette idée : pour construire la table d'analyse LL(1), il suffit de savoir calculer :

Les lexèmes
que le membre droit d'une règle peut produire en première position.

Un cas particulier

- ▶ Commençons par nous intéresser aux grammaires dont aucun non-terminal ne produit le mot vide.
- ▶ Cela signifie que pour tout membre droit α d'une règle, nous avons deux cas à considérer :
 - ▶ Soit $\alpha \equiv a\beta$ et le premier lexème produit par la règle est "a".
 - ▶ Soit $\alpha \equiv A\beta$ et l'ensemble des lexèmes produits en première position par la règle est le même que celui de A .

Équations entre ensemble de terminaux

- La définition précédente peut s'exprimer à l'aide de ces équations définissant $FIRST()$ où u est un mot de $(T \cup N)^+$:

$$\begin{aligned} FIRST(a) &= \{a\} \\ FIRST(A) &= FIRST(A) \\ FIRST(u) &= FIRST(A) \text{ Si } A \text{ dans } G \end{aligned}$$

Exercice

Comment calculer cette fonction $FIRST$?

Calcul de point fixe

- ▶ Une solution aux systèmes d'inéquations précédent est fournit par le plus petit point fixe de la fonction suivante :

$$\begin{aligned} FIRST^n(a) &= \{a\} \\ FIRST^n(A) &= FIRST^{n-1}(A) \\ FIRST^n(A) &= FIRST^{n-1}(A) \quad FIRST^{n-1}() \quad \text{Si } A \quad \text{dans } G \end{aligned}$$

- ▶ Pour calculer ce plus petit point fixe, on se donne une fonction $FIRST^0$ qui associe la fonction constante « w ».
- ▶ On itère ensuite la définition de cette fonction pour calculer une nouvelle fonction $FIRST^n$. Cette fonction associe à tous les mots des ensembles plus grands que la fonction de l'itération précédente.
- ▶ Or, la taille de ces ensembles est bornée par le cardinal de l'alphabet (fini) des non-terminaux.

Cette itération termine donc et est une solution du système d'inéquations.

Conflit FIRST/FIRST

- ▶ Si les ensembles FIRST de deux règles d'un non-terminal partagent un symbole, on dit que l'on a un **conflit FIRST/FIRST**.
- ▶ Une telle grammaire n'est donc pas LL(1).

Construction d'une table LL(1)

<i>Session</i>	::=	<i>Fact Session</i>
	/	<i>Question</i>
	/	<i>(Session)Session</i>
<i>Fact</i>	::=	<i>!STRING</i>
<i>Question</i>	::=	<i>?STRING</i>

Exercice

1. Est-ce que cette grammaire est adaptée à l'analyse LL ?
2. Calculer la fonction FIRST.
3. Calculer la table LL correspondante.
4. Analyser l'entrée « (!FIRE ?WATER) !WATER ?WOOD ».

Traitement des non-terminaux produisant le mot vide

- ▶ Face à une règle de la forme $A \rightarrow \epsilon$, on ne peut pas décider quels terminaux peuvent suivre A .
- ▶ Il faut aller voir les terminaux qui **peuvent suivre** A en observant les règles qui utilisent A .
- ▶ Pour effectuer cette inspection efficacement, on étend le co-domaine de la fonction $FIRST$: on rajoute un nouveau symbole ϵ à l'ensemble des terminaux.
- ▶ Si $\epsilon \in FIRST(A)$, cela signifie que le non terminal A peut produire le mot vide. On dit qu'il est **annulable**.
- ▶ On modifie aussi la définition de $FIRST$:

$$\begin{array}{ll} FIRST(\epsilon) &= \{\epsilon\} \\ FIRST(a) &= \{a\} \\ FIRST(A) &= FIRST(A) \quad \text{Si } \epsilon \notin FIRST(A) \\ FIRST(A) &= (FIRST(A) \setminus \{\epsilon\}) \cup \{\epsilon\} \quad \text{Si } \epsilon \in FIRST(A) \\ FIRST(A) &= FIRST(A) \quad \text{Si } A \text{ dans } G \end{array}$$

Un premier algorithme

- ▶ À l'aide de cette nouvelle version de la fonction FIRST, on peut définir un algorithme d'analyse syntaxique qui utilise la grammaire.
- ▶ À chaque étape de prédiction, si a est la prédiction courante, on calcule $FIRST(a)$ pour déterminer la règle de la grammaire à utiliser.
- ▶ Si il y a plusieurs règles, on échoue : la grammaire n'est pas LL(1).

Algorithme LL(1) en présence de règles

<i>Session</i>	$::=$	<i>Facts Question</i>
	$/$	<i>(Session)Session</i>
<i>Facts</i>	$::=$	<i>Fact Facts /</i>
<i>Fact</i>	$::=$	<i>!STRING</i>
<i>Question</i>	$::=$	<i>?STRING</i>

Exercice

1. Analysez l'entrée « !WATER (!FIRE ?WATER) ?WOOD » en calculant *FIRST* pour chaque étape.

Défauts de l'algorithme

- ▶ Le recalcul de *FIRST* à chaque étape de l'analyse est une source d'inefficacité en comparaison à l'analyse dirigée par une table.
- ▶ On peut cependant calculer une fonction *FOLLOW* pour chaque non-terminal A produisant le mot vide, qui représente l'ensemble des terminaux qui peuvent suivre A .
- ▶ Ainsi, pour construire la table, il suffit de rajouter chaque règle $A \rightarrow \epsilon$ dans toutes les cases (A, a) telles que $a \in FOLLOW(A)$.
- ▶ Si il existe un non terminal A pouvant produire le mot vide par une règle $A \rightarrow \epsilon$ et qu'il existe un terminal qui est à la fois dans $FIRST(A)$ et dans $FOLLOW(A)$ alors on a un conflit *FIRST/FOLLOW*.
- ▶ Si un non terminal possède deux règles distinctes de la forme $A \rightarrow \epsilon$ et $A \rightarrow \epsilon'$ qui peuvent produire le mot vide, alors il s'agit d'un conflit *FOLLOW/FOLLOW*.

Définition de la fonction

- ▶ On utilise une définition similaire à *FIRST* à l'aide d'inéquations :

$FIRST()$	$FOLLOW(A)$	Si $B \rightarrow A$ dans G .
$FOLLOW(B)$	$FOLLOW(A)$	Si $B \rightarrow A$ dans G et $FIRST()$.

- ▶ On peut encore résoudre ce système par itération du système récursif :

$$FOLLOW^n(A) = FOLLOW^{n-1}(\quad) \quad (G B \quad A \text{ dans })$$

Construction de la table LL(1) en présence de règles

<i>Session</i>	::=	<i>Facts Question</i>
	/	<i>(Session) Session</i>
<i>Facts</i>	::=	<i>Fact Facts /</i>
<i>Fact</i>	::=	<i>!STRING</i>
<i>Question</i>	::=	<i>?STRING</i>

Exercice

1. Calculer les fonctions FIRST et FOLLOW.
2. Calculer la table LL(1).
3. Analyser l'entrée « (!FIRE ?WATER) !WATER ?WOOD ».

Synthèse

Conclusion

- ▶ Nous avons présenté l'algorithme de Unger, très simple mais qui n'utilise pas l'entrée pour prédire les chemins de recherche les plus probables.
- ▶ Ceci nous a conduit à l'analyse prédictive, d'abord non déterministe, puis déterministe.
- ▶ La classe des grammaires LL(1) conduit à des analyseurs syntaxiques très efficaces mais certaines grammaires hors-contexte ne sont pas LL(1).
- ▶ Il y a des méthodes pour essayer de transformer certaines grammaires hors-contexte en grammaires LL(1).

INTRODUCTION À LA COMPILATION

Cours 4 : Analyse syntaxique ascendante

Yann Régis-Gianas
yrg@pps.univ-paris-diderot.fr

PPS - Université Denis Diderot – Paris 7

Exemple d'analyse ascendante

- Soit la grammaire (tirée de *Parsing Techniques* de Grune et Jacobs) :

(1)	$S \rightarrow aSQ$
(2)	$S \rightarrow abc$
(3)	$bQc \rightarrow bbcc$
(4)	$cQ \rightarrow Qc$

et l'entrée : « *aabbcc* »

a a b b c c

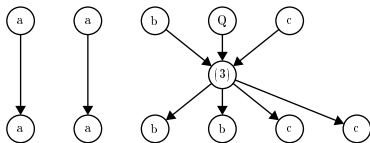
- Quelle règle peut produire une sous-séquence de non-terminaux de ce mot ?

Exemple d'analyse ascendante

- Soit la grammaire (tirée de *Parsing Techniques* de Grune et Jacobs) :

(1)	$S \rightarrow aSQ$
(2)	$S \rightarrow abc$
(3)	$bQc \rightarrow bbcc$
(4)	$cQ \rightarrow Qc$

et l'entrée : « *aabbcc* »



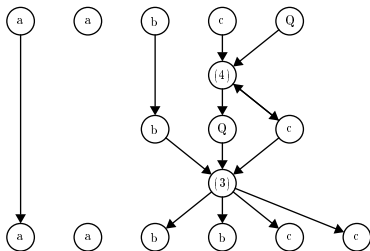
- De même, seule la règle (4) a pu produire le mot « *Q c* ».

Exemple d'analyse ascendante

- Soit la grammaire (tirée de *Parsing Techniques* de Grune et Jacobs) :

(1)	$S \rightarrow aSQ$
(2)	$S \rightarrow abc$
(3)	$bQc \rightarrow bbcc$
(4)	$cQ \rightarrow Qc$

et l'entrée : « *aabbcc* »



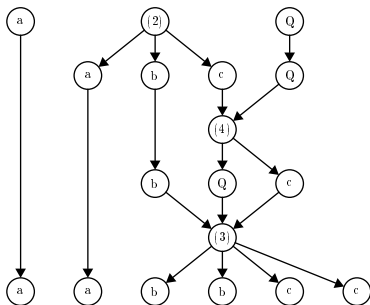
- Ici, seule la règle (3) s'applique et elle mène à l'entrée.

Exemple d'analyse ascendante

- Soit la grammaire (tirée de *Parsing Techniques* de Grune et Jacobs) :

(1)	$S \rightarrow aSQ$
(2)	$S \rightarrow abc$
(3)	$bQc \rightarrow bbcc$
(4)	$cQ \rightarrow Qc$

et l'entrée : « *aabbcc* »



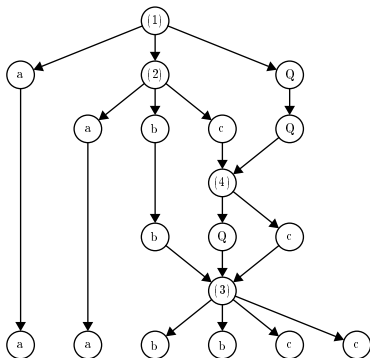
- Encore une fois, une seule règle s'applique, la règle (2).

Exemple d'analyse ascendante

Soit la grammaire (tirée de *Parsing Techniques* de Grune et Jacobs) :

(1)	$S \rightarrow aSQ$
(2)	$S \rightarrow abc$
(3)	$bQc \rightarrow bbcc$
(4)	$cQ \rightarrow Qc$

et l'entrée : « *aabbcc* »



Pour finir, seule la règle (1) produit le mot « *aSQ* ».

Algorithme CYK

Exemple de dérivation détaillée

- | Analysons le mot « $/aa/b/bb/$ » à l'aide de la grammaire G :

$$\begin{aligned} S &::= / L \\ L &::= \epsilon \mid E / L \\ E &::= A \mid B \\ A &::= aa \mid a \\ B &::= bb \mid b \end{aligned}$$

- On se concentre sur les non-terminaux qui produisent des sous-chaînes de taille 1.

/	a	a	/	b	/	b	b	/
---	---	---	---	---	---	---	---	---

Exemple de dérivation détaillée

- | Analysons le mot « $/aa/b/bb/$ » à l'aide de la grammaire G :

$$\begin{aligned} S &::= / L \\ L &::= \epsilon \mid E / L \\ E &::= A \mid B \\ A &::= aa \mid a \\ B &::= bb \mid b \end{aligned}$$

- On se concentre sur les non-terminaux qui produisent des sous-chaînes de taille 1.

	A	A		B		B	B	
$/$	a	a	$/$	b	$/$	b	b	$/$

Exemple de dérivation détaillée

- | Analysons le mot « $/aa/b/bb/$ » à l'aide de la grammaire G :

$$\begin{aligned} S &::= / L \\ L &::= \epsilon \mid E / L \\ E &::= A \mid B \\ A &::= aa \mid a \\ B &::= bb \mid b \end{aligned}$$

- Il faut prendre en compte les règles unitaires.

	E, A	E, A		E, B		E, B	E, B	
$/$	a	a	$/$	b	$/$	b	b	$/$

Exemple de dérivation détaillée

- Analysons le mot « $/aa/b/bb/$ » à l'aide de la grammaire G :

$$\begin{aligned} S &::= / L \\ L &::= \epsilon \mid E / L \\ E &::= A \mid B \\ A &::= aa \mid a \\ B &::= bb \mid b \end{aligned}$$

- On peut passer aux non-terminaux produisant des chaînes de taille 2.

	E, A					E, B		
	E, A	E, A		E, B		E, B	E, B	
$/$	a	a	$/$	b	$/$	b	b	$/$

Exemple de dérivation détaillée

- Analysons le mot « $/aa/b/bb/$ » à l'aide de la grammaire G :

$$\begin{aligned} S &::= / L \\ L &::= \epsilon \mid E / L \\ E &::= A \mid B \\ A &::= aa \mid a \\ B &::= bb \mid b \end{aligned}$$

- Pour reconnaître la fin de la chaîne comme une production de L , il faut « insérer » un mot vide à la fin de la chaîne.
- ⇒ En fait, il faudrait le faire entre tous les terminaux et au début de la chaîne.

					L			
					E, B			
	E, A							
	E, A	E, A		E, B		E, B	E, B	
$/$	a	a	$/$	b	$/$	b	b	$/ \quad \epsilon$

Exemple de dérivation détaillée

- Analysons le mot « $/aa/b/bb/$ » à l'aide de la grammaire G :

$$\begin{aligned} S &::= / L \\ L &::= \epsilon \mid E / L \\ E &::= A \mid B \\ A &::= aa \mid a \\ B &::= bb \mid b \end{aligned}$$

- On continue l'analyse en s'intéressant à des chaînes de plus en plus longues.

	L							
				L				
					L			
	E, A					E, B		
	E, A	E, A		E, B		E, B	E, B	
$/$	a	a	$/$	b	$/$	b	b	$/ \epsilon$

Exemple de dérivation détaillée

- Analysons le mot « $/aa/b/bb/$ » à l'aide de la grammaire G :

$$\begin{aligned} S &::= / L \\ L &::= \epsilon \mid E / L \\ E &::= A \mid B \\ A &::= aa \mid a \\ B &::= bb \mid b \end{aligned}$$

- L'analyse se termine lorsque l'on a reconnu l'ensemble du mot à analyser comme une production du symbole d'entrée de la grammaire.

S									
	L								
				L					
						L			
	E, A					E, B			
	E, A	E, A		E, B		E, B	E, B		
$/$	a	a	$/$	b	$/$	b	b	$/$	ϵ

Description formelle de l'algorithme (hors programme)

- | Soit R_ϵ , l'ensemble des non-terminaux qui peuvent produire le mot vide.
- | Soit $R_{i,l}$, l'ensemble des non-terminaux qui peuvent produire la sous-chaîne de s de longueur l à la position i , que nous noterons $S_{i,l}$.
(Si $s \equiv t_0 \dots t_n$ alors $S_{i,l} = t_i \dots t_{i+l-1}$.)
- | L'algorithme CYK est défini ainsi :

Pour l de 0 à n faire

Pour chaque $A \rightarrow c_1 \dots c_m$ de G faire

Pour i de 0 à $n - l$ faire

Si on peut partager $S_{i,l}$ en m segments $(w_i)_{i \in 1 \dots m}$

Tels que $\forall i, c_i$ produit w_i

Alors

$R_{i,l} \leftarrow R_{i,l} \cup \{A\}$

Fin

Fin

Fin

Étude de complexité (hors programme)

- | Soit n , la taille de l'entrée à analyser.
 - | Pour une règle dont le membre droit contient m symboles.
 - | On doit trouver $m - 1$ points de partitionnement de l'entrée.
 - | Une fois trouvé, un point de partitionnement doit être combiné avec tout ceux qui le précèdent.
 - | Trouver un point de partitionnement peut demander $O(n)$ actions.
 - | Trouver tous les points de partitionnement nécessite donc $O(n^{m-1})$ actions.
 - | Il y a $O(n^2)$ cases dans la table, l'algorithme demande donc $O(n^{m+1})$ actions.
- ⇒ Pour la grammaire précédente, l'analyse est donc de complexité $O(n^4)$.

CYK sur les grammaires en forme normale de Chomsky (hors programme)

- | Une grammaire est en forme normale de Chomsky si toutes ses règles sont de la forme :
 - ▶ « $A \rightarrow a$ » où a est un terminal.
 - ▶ « $A \rightarrow B C$ » où B et C sont des non-terminaux.
- ⇒ La complexité de CYK sur ces grammaires est “seulement” cubique.
- ⇒ L'absence de règles unitaires et de production vide rend inutiles les calculs de point fixe.

Mise en forme normale de Chomsky (hors programme)

- | Nous savons déjà supprimer les règles unitaires et de production vide.
- | Il reste alors des règles de la forme « $A \rightarrow c_1 \dots c_n$ » avec $n \geq 2$ et de la forme « $A \rightarrow a$ ».
- | Pour tout c_i qui est un terminal a_i , on introduit un non-terminal T_i ainsi qu'une règle « $T_i \rightarrow a_i$ ». On peut remplacer chaque occurrence de c_i dans les règles de la première forme par le non-terminal T_i correspondant. Le langage reconnu par la grammaire est bien sûr préservé.
- | Il reste alors des règles de la forme :

$$A \rightarrow A_1 \dots A_n$$

pour $n \geq 3$

- | Il suffit de les “découper” en multiples règles de la forme :

$$\begin{array}{lcl} A & \rightarrow & A_{1_n-1} \dots A_n \\ A_{1_n-1} & \rightarrow & A_{1_n-2} \dots A_{n-1} \\ & \dots & \end{array}$$

Exemple de mise en forme normale de Chomsky (hors programme)

$$\begin{aligned} S &::= / L \\ L &::= \epsilon \mid E / L \\ E &::= A \mid B \\ A &::= aa \mid a \\ B &::= bb \mid b \end{aligned}$$

Exercice

Mettre cette grammaire en forme normale de Chomsky.

Algorithme d'analyse ascendante dirigée

Une analyse directionnelle et ascendante

- | On choisit, encore une fois, de se contraindre à une lecture de gauche à droite du mot à analyser.
- | Sous cette contrainte, une analyse ascendante reconnaît nécessairement la dérivation **droite** dans l'ordre **inverse** de sa construction.

Une analyse directionnelle et ascendante

- | Voici un exemple :

$$\begin{aligned} S &::= A B C \\ A &::= a \\ B &::= b \\ C &::= c \end{aligned}$$

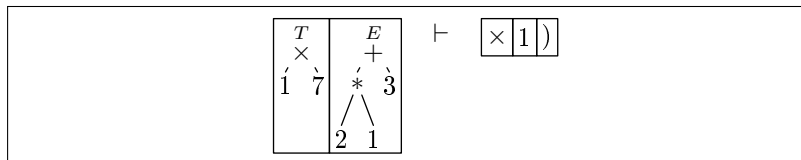
- | L'analyse ascendante du mot « a b c » suit les étapes suivantes (● représente le pointeur de lecture) :

1. ● a b c
2. A ● b c
3. A B ● c
4. A B C ●
5. S ●

- | En lisant cette séquence en sens inverse, on reconstruit bien la dérivation droite, c'est-à-dire celle qui réécrit le non-terminal le plus à droite.

Automate à pile pour l'analyse ascendante

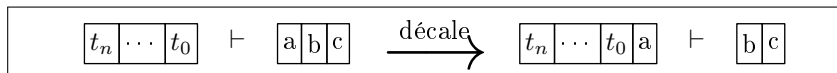
- Comme pour l'analyse descendante, l'automate à pile est un bon modèle de calcul pour l'analyse syntaxique ascendante.
- On représente une configuration d'analyse ascendante ainsi :



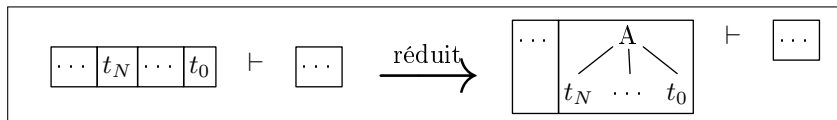
- À gauche du signe ' \vdash ' se trouve une **pile** des non-terminaux reconnus pour le moment avec leur arbre de production respectif. Le sommet de cette pile se trouve du côté du signe ' \vdash '.
- À droite du signe ' \vdash ' se trouve le reste de l'entrée à analyser.

Actions de l'automate à pile pour l'analyse ascendante

- | Il y a **deux** actions possibles : le **décalage** et la **réduction**.
- | Décaler, c'est lire la première lettre de l'entrée et la poser sur la pile :



- | Réduire, c'est dépiler N sous-arbres et empiler un nouvel arbre formé par l'application d'une règle de la grammaire :



- | Notez que ces deux opérations sont **inversibles**.

Espace de recherche

- | À tout moment, tant que l'entrée n'est pas consommée, l'automate peut effectuer une action « décale ».
- | À tout moment, un certain nombre, positif ou nul, d'actions « réduire » sont aussi possibles.
- | Une configuration est un échec, si aucune action n'est possible et que la pile n'est pas formée uniquement d'un arbre étiqueté par le non-terminal d'entrée de la grammaire.
- | Une configuration est un succès, si la totalité de l'entrée est consommée et que la pile est formée d'un unique arbre étiqueté par le non-terminal d'entrée de la grammaire.

⇒ Ces informations définissent un espace de recherche.

Utilisation d'un foncteur générique

- | Il existe une bibliothèque d'algorithmes de recherche écrite par J.C. Filliâtre : <http://www.lri.fr/~filliatr/ftp/ocaml/misc/search.ml.html>
- | Notre domaine de recherche peut être écrit sous la forme d'un module OCAML dont la signature est :

```
module type ImperativeProblem = sig  
  type move  
  val success : unit → bool  
  val moves : unit → move list  
  val domove : move → unit  
  val undomove : move → unit  
  val add : unit → unit  
  val mem : unit → bool  
  val clear : unit → unit  
end
```

Grammaire d'exemple

- I Nous allons analyser le mot « aaaab » par la grammaire suivante :

$$S ::= a N b$$
$$N ::= N a b$$
$$N ::= a a a$$

Algorithme de recherche en profondeur

- Voici la trace obtenue pour une recherche en **profondeur** :

<i>by shift</i>	$\vdash a a a a b$	$a \vdash a a a b$
<i>by shift</i>	$a \vdash a a a b$	$a a \vdash a a b$
<i>by shift</i>	$a a \vdash a a b$	$a a a \vdash a b$
<i>by shift</i>	$a a a \vdash a b$	$a a a a \vdash b$
<i>by shift</i>	$a a a a \vdash b$	$a a a a b \vdash$
<i>rolling back shift</i>	$a a a a b \vdash$	$a a a a \vdash b$
<i>by reducing top 3 elements using a a a to N</i>		
	$a a a a \vdash b$	$a (N a a a) \vdash b$
<i>by shift</i>	$a (N a a a) \vdash b$	$a (N a a a) b \vdash$
<i>by reducing top 3 elements using a N b to S</i>		
	$a (N a a a) b \vdash$	$(S a (N a a a) b) \vdash$

Algorithme de recherche en largeur

Voici la trace obtenue pour une recherche en **largeur** :

by shift	$\vdash a a a a b$	$a \vdash a a a b$
by shift	$a \vdash a a a b$	$a a \vdash a a b$
by shift	$a a \vdash a a b$	$a a a \vdash a b$
by shift	$a a a \vdash a b$	$a a a a \vdash b$
rolling back shift,	$a a a a \vdash b$	$a a a \vdash a b$
by reducing top 3 elements using $a a a$ to N	$a a a \vdash a b$	$(N a a a) \vdash a b$
rolling back reducing top 3 elements using $a a a$ to N	$(N a a a) \vdash a b$	$a a a \vdash a b$
by shift	$a a a \vdash a b$	$a a a a \vdash b$
by shift	$a a a a \vdash b$	$a a a a b \vdash$
rolling back shift,	$a a a a b \vdash$	$a a a a \vdash b$
by reducing top 3 elements using $a a a$ to N	$a a a a \vdash b$	$a (N a a a) \vdash b$
rolling back reducing top 3 elements using $a a a$ to N	$a (N a a a) \vdash b$	$a a a a \vdash b$
rolling back shift,	$a a a a \vdash b$	$a a a \vdash a b$
by reducing top 3 elements using $a a a$ to N	$a a a \vdash a b$	$(N a a a) \vdash a b$
by shift	$(N a a a) \vdash a b$	$(N a a a) a \vdash b$
rolling back shift,	$(N a a a) a \vdash b$	$(N a a a) a b \vdash$
rolling back reducing top 3 elements using $a a a$ to N	$(N a a a) a \vdash b$	$(N a a a) a \vdash b$
rolling back shift,	$(N a a a) a \vdash b$	$(N a a a) \vdash a b$
rolling back reducing top 3 elements using $a a a$ to N	$(N a a a) \vdash a b$	$a a a a \vdash b$
by shift	$a a a a \vdash b$	$a a a a \vdash b$
by reducing top 3 elements using $a a a$ to N	$a a a a \vdash b$	$a (N a a a) \vdash b$
by shift	$a (N a a a) \vdash b$	$a (N a a a) b \vdash$
rolling back shift,	$a (N a a a) b \vdash$	$a (N a a a) \vdash b$

rolling back reducing top 3 elements using $a a a$ to N	$a (N a a a) \vdash b$	$a a a a \vdash b$
rolling back shift,	$a a a a \vdash b$	$a a a \vdash a b$
by reducing top 3 elements using $a a a$ to N	$a a a \vdash a b$	$(N a a a) \vdash a b$
by shift	$(N a a a) \vdash a b$	$(N a a a) a \vdash b$
by shift	$(N a a a) a \vdash b$	$(N a a a) a b \vdash$
rolling back shift,	$(N a a a) a b \vdash$	$(N a a a) a \vdash b$
rolling back shift,	$(N a a a) a \vdash b$	$(N a a a) \vdash a b$
rolling back reducing top 3 elements using $a a a$ to N	$(N a a a) \vdash a b$	$a a a \vdash a b$
by shift	$a a a \vdash a b$	$a a a a \vdash b$
by reducing top 3 elements using $a a a$ to N	$a a a a \vdash b$	$a (N a a a) \vdash b$
by shift	$a (N a a a) \vdash b$	$a (N a a a) b \vdash$
by reducing top 3 elements using $a N b$ to S	$a (N a a a) b \vdash$	$(S a (N a a a) b) \vdash$

Grammaire d'exemple plus réaliste

- Combien d'étapes sont nécessaires à l'analyse du mot « a - a + a » par la grammaire suivante ?

$$\begin{aligned} S &::= E \\ E &::= E Q F \mid F \\ F &::= a \\ Q &::= + \mid - \end{aligned}$$

⇒ 292 pour la recherche en profondeur, 1369 pour la recherche en largeur !

Algorithme de Earley

Présentation

- | L'algorithme de Earley coupe des branches de l'arbre de recherche en prenant en compte le **contexte** dans lequel l'analyse ascendante est effectuée.
- | Dans l'exemple précédent :

<i>by shift</i>	$\vdash a - a + a$	$a \vdash - a + a$
<i>by shift</i>	$a \vdash - a + a$	$a - \vdash a + a$
<i>by shift</i>	$a - \vdash a + a$	$a - a \vdash + a$
<i>by shift</i>	$a - a \vdash + a$	$a - a + \vdash a$
<i>by shift</i>	$a - a + \vdash a$	$a - a + a \vdash$
<i>by reducing top 1 elements using a to F</i>	$a - a + a \vdash$	$a - a + (F a) \vdash$
<i>by reducing top 1 elements using F to E</i>	$a - a + (F a) \vdash$	$a - a + (E (F a)) \vdash$
<i>by reducing top 1 elements using E to S</i>	$a - a + (E (F a)) \vdash$	$a - a + (S (E (F a))) \vdash$

- | Après avoir vu passer le terminal '-', on peut initier la reconnaissance d'un non-terminal F mais on sait qu'un E et un S ne peuvent pas être reconnus.
- ⇒ L'analyse ascendante doit être dirigée par une analyse descendante.

Les “états d’analyse” de l’algorithme de Earley

- | L’algorithme de Earley utilise une description de l’état de l’analyse, appelé *item* en anglais, qui suit la syntaxe suivante :

$$N \rightarrow s_1 \dots s_k \bullet s'_1 \dots s'_n$$

où les s_i et les s'_j sont des symboles terminaux ou non-terminaux.

- | Cette description se lit :
« *Au sein de l’analyse du non-terminal N , les symboles s_1, \dots, s_k ont été reconnus et les symboles s'_1, \dots, s'_n sont attendus par la suite (ils sont prédits par l’analyse).* »

Classification des états de l'analyse

- | La position du symbole \bullet caractérise différents stades de l'analyse.
- | **État à réduire** « $N \rightarrow s_1 \dots s_k \bullet$ » :
L'analyse de N est terminée. On peut légitimement réduire les k éléments du sommet de la pile en utilisant cette règle. L'analyse peut se poursuivre dans l'état qui a initié l'analyse du non-terminal N .
- | **État prédit** « $N \rightarrow \bullet s'_1 \dots s'_n$ » :
L'analyse de N débute. Cet état a été initié par une prédiction.
- | **État à décaler** « $N \rightarrow s_1 \dots s_k \bullet a \dots s'_n$ » :
L'analyse prédit un terminal. L'automate peut seulement effectuer l'action « décale ».
- | **État de prédiction** « $N \rightarrow s_1 \dots s_k \bullet N' \dots s'_n$ » :
L'analyse prédit un non-terminal. On doit considérer l'ensemble des états prédits qui ont un sens en ce point.

Description étendue d'état d'analyse

- | Pour être capable d'associer un état prédit à son état de prédiction, on utilise la **position dans le mot à analyser** correspondant à la position du non-terminal reconnu par l'état prédit.
- | En d'autres termes, on associe un entier à chaque description d'état d'analyse qui correspond à la position à partir de laquelle cette analyse a été initiée.
- | On note ainsi la description étendue que l'on obtient :

$$N \rightarrow s_1 \dots s_k \bullet s'_1 \dots s'_n \quad @ p$$

- | Elle se lit :
*« Au sein de l'analyse du non-terminal **N** initiée à la position **p** de l'entrée, les symboles s_1, \dots, s_k ont été reconnus et les symboles s'_1, \dots, s'_n sont attendus par la suite (on dit aussi qu'ils sont prédits par l'analyse). »*

L'algorithme de Earley (version naïve) (hors programme)

- | L'algorithme de Earley est décomposable en trois fonctions.
- | Ces fonctions travaillent sur quatre **ensembles de descriptions d'état d'analyse** :
 - ▶ Complétés_p : l'ensemble des états à réduire à la position p .
 - ▶ Actifs_p : l'ensemble des états à décaler ou en prédiction à la position p .
 - ▶ Prédits_p : l'ensemble des états prédits à la position p .
 - ▶ Tous_p : tous les états de la position p .

L'algorithme de Earley (version naïve) (hors programme)

- | L'algorithme de Earley est décomposable en trois fonctions.
- | **Fonction de lecture de l'entrée « scan » :**
Transforme les états actifs à décaler à la position p en états de la position $p + 1$. Ces états peuvent être à réduire, à décaler ou de prédiction.
- | **Fonction de réduction « complete » :**
Traite les états complétés à la position p dont l'analyse a été initiée en m . Appelle la fonction `scan` sur tous les états actifs en position m qui ont initié l'analyse du non-terminal reconnu en p . Cette fonction décale ces états en $p + 1$. (Plusieurs itérations peuvent être nécessaires.)
- | **Fonction de prédiction « predict » :**
Pour chaque état de prédiction actif en p dont le non-terminal prédit en tête est N , rajoute dans Prédits_p les états « $N \rightarrow \bullet s_1 \dots s_n @ p$ » pour chaque règle de la forme « $N \rightarrow s_1 \dots s_n$ » dans la grammaire. (Plusieurs itérations peuvent être nécessaires.)

Exemple

- | Analysons le mot “a - a + a” à travers la grammaire :

$$\begin{aligned} S &::= E \\ E &::= E Q F \mid F \\ F &::= a \\ Q &::= + \mid - \end{aligned}$$

Exemple (hors programme)

$$\begin{array}{c} \text{Complet}s_1 \\ \emptyset \\ \text{Actifs}_1 \cup \text{Predits}_1 \\ S \rightarrow \bullet E @ 1 \end{array} a \begin{array}{c} \text{Complet}s_2 \\ \text{Actifs}_2 \cup \text{Predits}_2 \end{array} - \begin{array}{c} \text{Complet}s_3 \\ \text{Actifs}_3 \cup \text{Predits}_3 \end{array} a \begin{array}{c} \text{Complet}s_4 \\ \text{Actifs}_4 \cup \text{Predits}_4 \end{array} + \begin{array}{c} \text{Complet}s_5 \\ \text{Actifs}_5 \cup \text{Predits}_5 \end{array} a$$

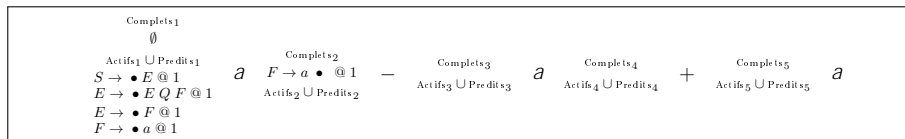
- Les règles du symbole d'entrée produisent les états d'analyse actifs initiaux.

Exemple (hors programme)

$$\begin{array}{l}
 \text{Complet}s_1 \\
 \emptyset \\
 \text{Actifs}_1 \cup \text{Predits}_1 \\
 S \rightarrow \bullet E @ 1 \\
 E \rightarrow \bullet E Q F @ 1 \\
 F \rightarrow \bullet F @ 1 \\
 F \rightarrow \bullet a @ 1
 \end{array}
 a
 -
 \begin{array}{l}
 \text{Complet}s_2 \\
 \text{Actifs}_2 \cup \text{Predits}_2
 \end{array}
 +
 \begin{array}{l}
 \text{Complet}s_3 \\
 \text{Actifs}_3 \cup \text{Predits}_3
 \end{array}
 a
 +
 \begin{array}{l}
 \text{Complet}s_4 \\
 \text{Actifs}_4 \cup \text{Predits}_4
 \end{array}
 +
 \begin{array}{l}
 \text{Complet}s_5 \\
 \text{Actifs}_5 \cup \text{Predits}_5
 \end{array}
 a$$

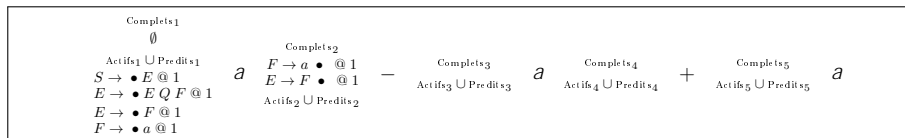
- La fonction `predict` rajoute toutes les expansions possibles de E .

Exemple (hors programme)



- La fonction scan décale le terminal en tête a et insère l'état d'analyse obtenu dans l'ensemble Complets_2 .

Exemple (hors programme)



- La fonction `complete` réduit le terminal a sur la pile en un non-terminal F puis appelle la fonction `SCAN` pour qu'elle tienne compte de cette réduction.

Exemple (hors programme)

$$\begin{array}{c}
 \text{Complet}s_1 \\
 \emptyset \\
 \text{Actifs}_1 \cup \text{Predits}_1 \\
 S \rightarrow \bullet E @ 1 \\
 E \rightarrow \bullet E Q F @ 1 \\
 E \rightarrow \bullet F @ 1 \\
 F \rightarrow \bullet a @ 1
 \end{array}
 a
 \begin{array}{c}
 \text{Complet}s_2 \\
 F \rightarrow a \bullet @ 1 \\
 E \rightarrow F \bullet @ 1 \\
 S \rightarrow E \bullet @ 1 \\
 \text{Actifs}_2 \cup \text{Predits}_2 \\
 E \rightarrow E \bullet Q F @ 1
 \end{array}
 -
 \begin{array}{c}
 \text{Complet}s_3 \\
 \text{Actifs}_3 \cup \text{Predits}_3
 \end{array}
 a
 \begin{array}{c}
 \text{Complet}s_4 \\
 \text{Actifs}_4 \cup \text{Predits}_4
 \end{array}
 +
 \begin{array}{c}
 \text{Complet}s_5 \\
 \text{Actifs}_5 \cup \text{Predits}_5
 \end{array}
 a$$

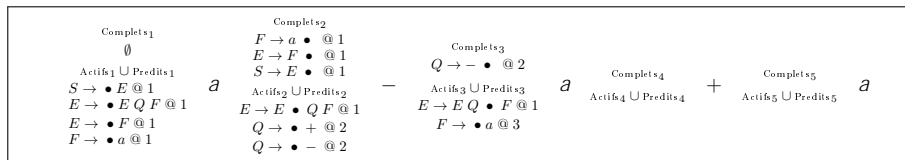
- Par itération de cette interaction entre la fonction scan et complète, on complète l'ensemble Tous₂.

Exemple (hors programme)

$$\begin{array}{c}
 \text{Complet}s_1 \\
 \emptyset \\
 \text{Actifs}_1 \cup \text{Predits}_1 \\
 S \rightarrow \bullet E @ 1 \\
 E \rightarrow \bullet E Q F @ 1 \\
 E \rightarrow \bullet F @ 1 \\
 F \rightarrow \bullet a @ 1
 \end{array}
 a
 \begin{array}{c}
 \text{Complet}s_2 \\
 F \rightarrow a \bullet @ 1 \\
 E \rightarrow F \bullet @ 1 \\
 S \rightarrow E \bullet @ 1 \\
 \text{Actifs}_2 \cup \text{Predits}_2 \\
 E \rightarrow E \bullet Q F @ 1 \\
 Q \rightarrow \bullet + @ 2 \\
 Q \rightarrow \bullet - @ 2
 \end{array}
 -
 \begin{array}{c}
 \text{Complet}s_3 \\
 \text{Actifs}_3 \cup \text{Predits}_3
 \end{array}
 a
 \begin{array}{c}
 \text{Complet}s_4 \\
 \text{Actifs}_4 \cup \text{Predits}_4
 \end{array}
 +
 \begin{array}{c}
 \text{Complet}s_5 \\
 \text{Actifs}_5 \cup \text{Predits}_5
 \end{array}
 a$$

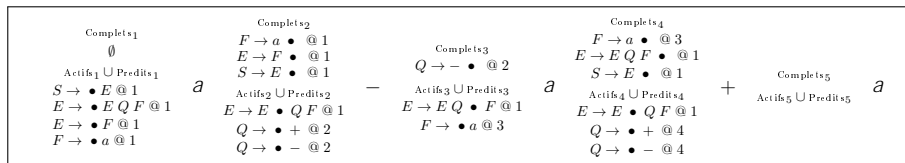
- Il apparaît alors de nouveaux états de prédiction à traiter par la fonction `predict`.

Exemple (hors programme)



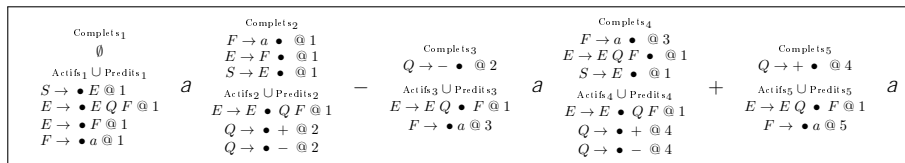
- | On suit ce procédé pour traiter l'ensemble du mot à analyser.
- | À la fin de la lecture du mot d'entrée, on doit trouver un état à réduire vers le symbole d'entrée de la grammaire.

Exemple (hors programme)



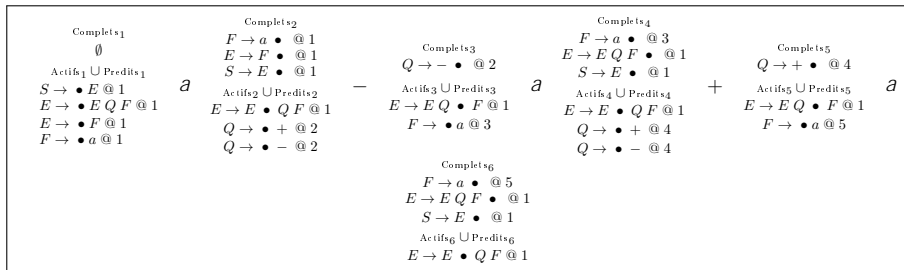
- | On suit ce procédé pour traiter l'ensemble du mot à analyser.
- | À la fin de la lecture du mot d'entrée, on doit trouver un état à réduire vers le symbole d'entrée de la grammaire.

Exemple (hors programme)



- | On suit ce procédé pour traiter l'ensemble du mot à analyser.
- | À la fin de la lecture du mot d'entrée, on doit trouver un état à réduire vers le symbole d'entrée de la grammaire.

Exemple (hors programme)



- | On suit ce procédé pour traiter l'ensemble du mot à analyser.
- | À la fin de la lecture du mot d'entrée, on doit trouver un état à réduire vers le symbole d'entrée de la grammaire.

Problème des règles de production vide (hors programme)

- | Encore une fois, les règles de production vide posent problème.
- | Analysons le mot « aa / a » à travers la grammaire suivante :

$$\begin{aligned} S &::= E \\ E &::= E Q F \mid F \\ F &::= a \\ Q &::= \times \mid / \mid \epsilon \end{aligned}$$

Exemple problématique (hors programme)

$$\begin{array}{c} \text{Complet s}_1 \\ \emptyset \\ \text{Actifs}_1 \cup \text{Predits}_1 \\ S \rightarrow \bullet E @ 1 \end{array} a \quad \begin{array}{c} \text{Complet s}_2 \\ \text{Actifs}_2 \cup \text{Predits}_2 \end{array} a \quad \begin{array}{c} \text{Complet s}_4 \\ \text{Actifs}_4 \cup \text{Predits}_4 \end{array} = \begin{array}{c} \text{Complet s}_5 \\ \text{Actifs}_5 \cup \text{Predits}_5 \end{array} a$$

- | Les premières étapes ne posent pas de problème.

Exemple problématique (hors programme)

$$\begin{array}{l} \text{Complet s}_1 \\ \emptyset \\ \text{Actifs}_1 \cup \text{Predits}_1 \\ S \rightarrow \bullet E @ 1 \\ E \rightarrow \bullet E Q F @ 1 \\ E \rightarrow \bullet F @ 1 \\ F \rightarrow \bullet a @ 1 \end{array} a \quad \begin{array}{l} \text{Complet s}_2 \\ \text{Actifs}_2 \cup \text{Predits}_2 \end{array} a \quad \begin{array}{l} \text{Complet s}_4 \\ \text{Actifs}_4 \cup \text{Predits}_4 \end{array} = \begin{array}{l} \text{Complet s}_5 \\ \text{Actifs}_5 \cup \text{Predits}_5 \end{array} a$$

- | Les premières étapes ne posent pas de problème.

Exemple problématique (hors programme)

$$\begin{array}{c}
 \text{Complets}_1 \\
 \emptyset \\
 \text{Actifs}_1 \cup \text{Predits}_1 \\
 S \rightarrow \bullet E @ 1 \\
 E \rightarrow \bullet E Q F @ 1 \\
 E \rightarrow \bullet F @ 1 \\
 F \rightarrow \bullet a @ 1
 \end{array}
 \quad a \quad
 \begin{array}{c}
 \text{Complets}_2 \\
 F \rightarrow a \bullet @ 1 \\
 E \rightarrow F \bullet @ 1 \\
 S \rightarrow E \bullet @ 1 \\
 Q \rightarrow \bullet @ 1 \\
 \text{Actifs}_2 \cup \text{Predits}_2 \\
 E \rightarrow E \bullet Q F @ 1 \\
 Q \rightarrow \bullet * @ 2 \\
 Q \rightarrow \bullet / @ 2 \\
 F \rightarrow \bullet a @ 2
 \end{array}
 \quad a \quad
 \begin{array}{c}
 \text{Complets}_4 \\
 \text{Actifs}_4 \cup \text{Predits}_4
 \end{array}
 =
 \begin{array}{c}
 \text{Complets}_5 \\
 \text{Actifs}_5 \cup \text{Predits}_5
 \end{array}
 \quad a$$

- | La fonction `predict` doit rajouter l'état « $Q \rightarrow \epsilon \bullet$ » à l'ensemble Complets_2 .
- | Or, cet ensemble est censé être déjà déterminé par la fonction `complete`.
- ⇒ On itère une alternance d'application des fonctions `complete` et `predict`.
- ⇒ Heureusement, ce processus termine. (Pourquoi ?)

Exemple problématique (hors programme)

Complets_1 \emptyset $\text{Actifs}_1 \cup \text{Predits}_1$ $S \rightarrow \bullet E @ 1$ $E \rightarrow \bullet E Q F @ 1$ $E \rightarrow \bullet F @ 1$ $F \rightarrow \bullet a @ 1$	a	Complets_2 $F \rightarrow a \bullet @ 1$ $E \rightarrow F \bullet @ 1$ $S \rightarrow E \bullet @ 1$ $Q \rightarrow \bullet @ 1$	a	Complets_4 $F \rightarrow a \bullet @ 2$ $E \rightarrow E Q F \bullet @ 1$ $S \rightarrow E \bullet @ 1$ $Q \rightarrow \bullet @ 3$	$=$	Complets_5 $Q \rightarrow / \bullet @ 3$ $\text{Actifs}_5 \cup \text{Predits}_5$ $E \rightarrow E Q \bullet F @ 1$ $F \rightarrow \bullet a @ 4$	a	Complets_5 $F \rightarrow a \bullet @ 4$ $E \rightarrow E Q F \bullet @ 1$ $S \rightarrow E \bullet @ 1$ $Q \rightarrow \bullet @ 5$
		$\text{Actifs}_2 \cup \text{Predits}_2$ $E \rightarrow E \bullet Q F @ 1$ $\triangleright E \rightarrow E \bullet Q F @ 1$ $Q \rightarrow \bullet * @ 2$ $Q \rightarrow \bullet / @ 2$ $F \rightarrow \bullet a @ 2$		$\text{Actifs}_4 \cup \text{Predits}_4$ $E \rightarrow E \bullet Q F @ 1$ $\triangleright E \rightarrow E Q \bullet F @ 1$ $Q \rightarrow \bullet * @ 3$ $Q \rightarrow \bullet / @ 3$ $F \rightarrow \bullet a @ 3$				$\text{Actifs}_5 \cup \text{Predits}_5$ $E \rightarrow E \bullet Q F @ 1$ $\triangleright E \rightarrow E Q \bullet F @ 1$ $Q \rightarrow \bullet * @ 5$ $Q \rightarrow \bullet / @ 5$ $F \rightarrow \bullet a @ 5$

- La fonction `predict` doit rajouter l'état « $Q \rightarrow \epsilon \bullet$ » à l'ensemble `Complets2`.
 - Or, cet ensemble est censé être déjà déterminé par la fonction `complete`.
- ⇒ On itère une alternance d'application des fonctions `complete` et `predict`.
- ⇒ Heureusement, ce processus termine. (Pourquoi ?)

Suppression de l'itération complete/predict (hors programme)

- | On peut se passer de l'itération précédente en réorganisant l'algorithme et en modifiant légèrement la fonction `predict`.
- | On calcule, pour chaque position p , une liste ℓ_p définie à l'aide de ℓ_{p-1} .
- | Initialement, ℓ_p contient les états d'analyse insérés par la fonction `scan` à l'itération précédente.
- | Pour chaque état de la liste ℓ_p , si :
 - ▶ c'est un état à décaler, alors on applique la fonction `scan`.
 - ▶ c'est un état à réduire, alors on applique la fonction `complete`.
 - ▶ c'est un état de prédiction, alors on applique la fonction `predict`.

⇒ Si ces deux dernières fonctions doivent insérer un nouvel état à la position p , cet état est rajouté **à la fin de** ℓ_p .
- | La fonction `predict`, confrontée à un état d'analyse de la forme « $N \rightarrow \dots \bullet B \dots$ » où B peut produire le mot vide, produit, en plus des prédictions d'expansion de B , un état « $N \rightarrow \dots B \bullet \dots$ » inséré à la fin de ℓ_p .

Équivalence entre les deux algorithmes (hors programme)

- | La preuve d'équivalence entre les deux algorithmes, l'un utilisant une itération et l'autre non, n'est pas immédiate.
- | En fait, dans le second algorithme, l'itération est déplacée dans la fonction de calcul des non-terminaux produisant le mot vide.

Étude de complexité (hors programme)

Exercice

Montrer que l'algorithme de Earley est cubique, en pire cas.

Dirigée la prédiction par le prochain terminal de l'entrée

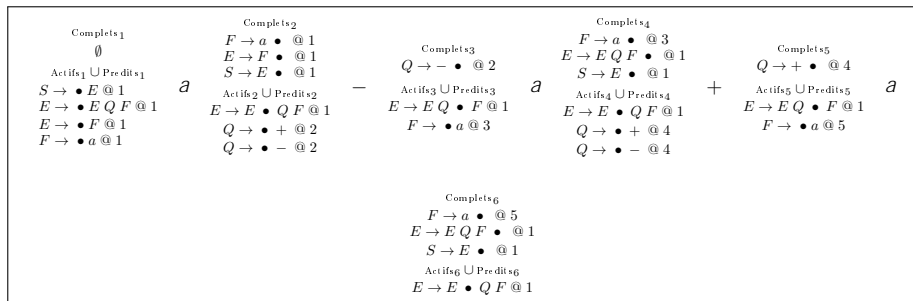
(hors programme)

- Comme dans le cas des analyses ascendantes, on remarque qu'une grande partie des prédictions peut être éliminée par une observation du terminal suivant de l'entrée.
- À l'étape suivante de notre exemple :

$$\begin{array}{c} \text{Complet}s_1 \\ \emptyset \\ \text{Actifs}_1 \cup \text{Predits}_1 \\ S \rightarrow \bullet E @ 1 \\ E \rightarrow \bullet E Q F @ 1 \\ E \rightarrow \bullet F @ 1 \\ F \rightarrow \bullet a @ 1 \end{array} a \quad \begin{array}{c} \text{Complet}s_2 \\ F \rightarrow a \bullet @ 1 \\ E \rightarrow F \bullet @ 1 \\ S \rightarrow E \bullet @ 1 \\ \text{Actifs}_2 \cup \text{Predits}_2 \\ E \rightarrow E \bullet Q F @ 1 \\ Q \rightarrow \bullet + @ 2 \\ Q \rightarrow \bullet - @ 2 \end{array} - \quad \begin{array}{c} \text{Complet}s_3 \\ \text{Actifs}_3 \cup \text{Predits}_3 \end{array} a \quad \begin{array}{c} \text{Complet}s_4 \\ \text{Actifs}_4 \cup \text{Predits}_4 \end{array} + \quad \begin{array}{c} \text{Complet}s_5 \\ \text{Actifs}_5 \cup \text{Predits}_5 \end{array} a$$

- La prédiction « $Q \rightarrow \bullet + @ 2$ » est idiote car le prochain terminal est "-".
- ⇒ On peut généraliser cette idée en utilisant l'ensemble *FIRST*.

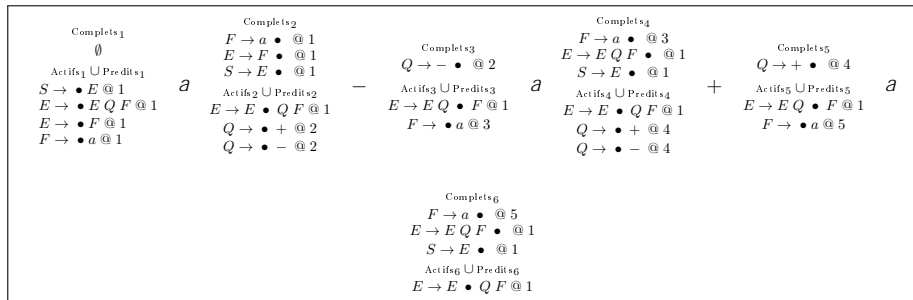
Application (hors programme)



Exercice

Quelles sont les prédictions idiotes ?

Encore une amélioration ? (hors programme)



Exercice

Quels sont les états à réduire éliminables par observation du prochain terminal ?

Dirigée la réduction par le prochain terminal de l'entrée

(hors programme)

- | Pour l'analyse LL(1), nous avons calculé l'ensemble $FOLLOW(N)$ des terminaux qui peuvent suivre le non-terminal N dans l'entrée.
- | Nous pouvons réutiliser cette information pour étendre de nouveau la description des états d'analyse en lui rajoutant un ensemble de **terminaux suivants potentiels** :

$$N \rightarrow s_1 \dots s_k \bullet s'_1 \dots s'_n \quad [a_1 \dots a_m]@p$$

avec $FOLLOW(N) = \{a_1 \dots a_m\}$

- | Il ne sert à rien de réduire un état de $Complets_p$ si le prochain terminal n'est pas dans son ensemble de terminaux suivants potentiels.

Trace finale de l'algorithme de Earley (hors programme)

Complets₁
 \emptyset
 Actifs₁ \cup Predits₁
 $S \rightarrow \bullet E [\#] @ 1$
 $E \rightarrow \bullet E Q F [\# + -] @ 1$
 $E \rightarrow \bullet F [\# + -] @ 1$
 $F \rightarrow \bullet a [\# + -] @ 1$

a

Complets₂
 $F \rightarrow a \bullet [\# + -] @ 1$
 $E \rightarrow F \bullet [\# + -] @ 1$
 $S \rightarrow E \bullet [\#] @ 1$
 Actifs₂ \cup Predits₂
 $E \rightarrow E \bullet Q F [\# + -] @ 1$
 $Q \rightarrow \bullet + [a] @ 2$
 $Q \rightarrow \bullet - [a] @ 2$

-

Complets₃
 $Q \rightarrow - \bullet [a] @ 2$
 Actifs₃ \cup Predits₃
 $E \rightarrow E Q \bullet F [\# + -] @ 1$
 $F \rightarrow \bullet a [\# + -] @ 3$

a

Complets₄
 $F \rightarrow a \bullet [\# + -] @ 3$
 $E \rightarrow E Q F \bullet [\# + -] @ 1$
 $S \rightarrow E \bullet [\#] @ 1$
 Actifs₄ \cup Predits₄
 $E \rightarrow E \bullet Q F [\# + -] @ 1$
 $Q \rightarrow \bullet + [a] @ 4$
 $Q \rightarrow \bullet - [a] @ 4$

+

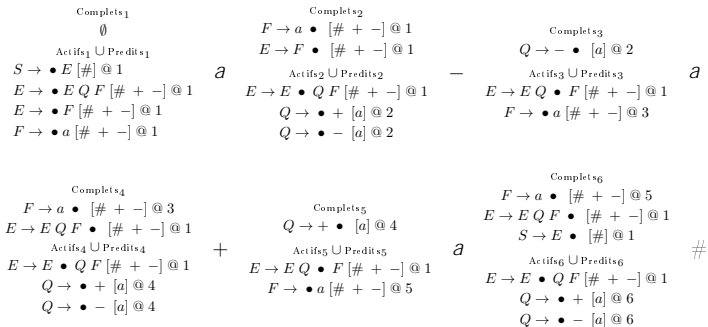
Complets₅
 $Q \rightarrow + \bullet [a] @ 4$
 Actifs₅ \cup Predits₅
 $E \rightarrow E Q \bullet F [\# + -] @ 1$
 $F \rightarrow \bullet a [\# + -] @ 5$

a

Complets₆
 $F \rightarrow a \bullet [\# + -] @ 5$
 $E \rightarrow E Q F \bullet [\# + -] @ 1$
 $S \rightarrow E \bullet [\#] @ 1$
 Actifs₆ \cup Predits₆
 $E \rightarrow E \bullet Q F [\# + -] @ 1$
 $Q \rightarrow \bullet + [a] @ 6$
 $Q \rightarrow \bullet - [a] @ 6$

#

Trace finale de l'algorithme de Earley (hors programme)



Synthèse

Synthèse

- | L'algorithme CYK nécessite une grammaire en forme normale de Chomsky pour effectuer une analyse syntaxique en $O(n^3)$.
 - | Les deux actions d'un automate à pile d'analyse ascendante sont « décale » et « réduit ». Elles forment un espace de recherche que l'on peut parcourir en temps exponentiel.
 - | L'algorithme de Earley élague cet arbre de recherche en étant dirigé par une analyse descendante parallèle. Il effectue ainsi une analyse syntaxique en $O(n^3)$.
- ⇒ La prochaine fois, pour le dernier cours sur l'analyse syntaxique, nous étudierons les algorithmes de la famille LR, qui sont linéaires et utilisés par les outils de type YACC tels que MENHIR.

QCM : Question 1

Dans l'algorithme CYK, la case (i, l) de la table contient :

- ☐ le sous-mot de longueur l à la position i .
- ☐ le non-terminal qui peut produire le sous-mot de longueur l à la position i .
- ☐ l'ensemble des non-terminaux qui peuvent produire le sous-mot de longueur l à la position i .
- ☐ la pile de l'automate utilisé pour analyser le sous-mot à la position i et de longueur l .

QCM : Question 2

La complexité en pire cas de l'algorithme CYK est

- ☐ $O(n^4)$
- ☐ $O(n^3)$
- ☐ $O(n^{m+1})$ avec m la longueur maximale des règles de la grammaire.
- ☐ $O(n)$

QCM : Question 3

À propos des grammaires de Chomsky, quelles affirmations sont vraies :

- ☐ On peut associer une forme normale de Chomsky à toute grammaire hors-contexte.
- ☐ L'algorithme CYK est quadratique sur les grammaires de Chomsky.
- ☐ Une grammaire de Chomsky peut contenir des règles unitaires.
- ☐ La forme normale de Chomsky d'une grammaire G a strictement plus de règles que G .

QCM : Question 5

Dans un automate à pile d'analyse ascendante,

- ☐ on décale en dépilant le prochain terminal à traiter de la pile.
- ☐ on décale en empilant le prochain terminal sur la pile.
- ☐ on réduit en lisant N terminaux d'entrée.
- ☐ on réduit en dépilant N symboles et en appliquant une règle de la grammaire dont on empile le résultat.
- ☐ il peut y avoir une séquence de transitions qui produit une analyse qui ne termine pas.

QCM : Question 6

Un parcours en profondeur :

- ☐ trouve plus rapidement une solution qu'un autre parcours.
- ☐ consomme plus de mémoire qu'un autre parcours.
- ☐ permet de ne lire l'entrée qu'une seule fois.
- ☐ a une complexité linéaire en la taille de l'entrée.

QCM : Question 7

L'algorithme de Earley :

- ☐ utilise une fonction de prédiction qui peut lire un terminal en avance.
- ☐ est exponentiel.
- ☐ ne traite pas les grammaires avec règles unitaires.
- ☐ fonctionne sur l'ensemble des grammaires hors-contextes.

Introduction à la Compilation

Cours 5 : Analyse syntaxique ascendante (deuxième partie)

Yann Régis-Gianas
`yrg@pps.univ-paris-diderot.fr`

PPS - Université Denis Diderot – Paris 7

Vers une analyse ascendante déterministe

Retour sur l'algorithme de Earley

- | L'algorithme de Earley améliore l'analyse ascendante naïve en la dirigeant à l'aide d'une analyse descendante.
 - | L'analyse descendante est modélisée par les ensembles d'états d'analyse.
 - | Ces états correspondent à plusieurs analyses **concurrentes**.
 - | Le traitement de ces analyses concurrentes peut mener à une exécution en temps cubique.
 - | En s'intéressant aux grammaires pour lesquelles il n'y a qu'une seule analyse possible à la fois, on peut s'assurer d'une complexité linéaire.
- ⇒ Ce sont les algorithmes de la famille **LR** dont nous allons parlés aujourd'hui.

Algorithme LR(0)

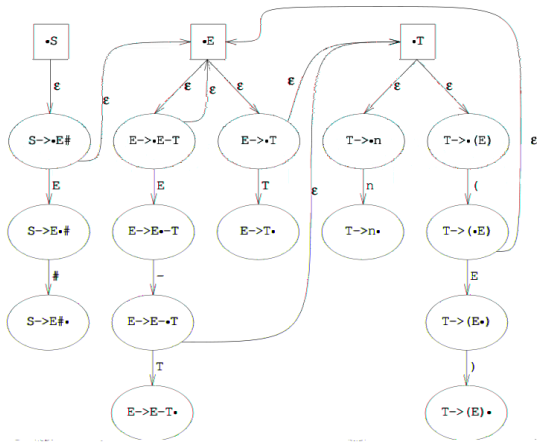
Un automate qui prédit le futur état de l'analyse

- | Dans l'algorithme de Earley, plusieurs non-terminaux pouvaient être en cours d'analyse. En effet, la fonction `predict` introduit autant d'états d'analyse qu'il y a de règles définissant le non-terminal considéré.
- | On peut formuler ce comportement à l'aide d'un **automate fini non-déterministe** qui **analyse la phrase intermédiaire reconnue jusqu'à maintenant** pour déterminer le **prochain état d'analyse**.

Un automate qui prédit le futur état de l'analyse

- | Pour chaque état de l'analyse, on veut trouver un état dans l'automate dont chaque transition est étiquetée par le symbole à reconnaître pour passer de cet état à un état d'analyse suivant.
- | De façon à simplifier la formulation de l'automate, pour chaque non-terminal N , on définit un état « gare » représentant le début de l'analyse de ce non-terminal, que l'on note « $\bullet N$ »
- | Ainsi, si on a un état d'analyse de la forme « $E \rightarrow \dots \bullet F \dots$ », on insère une transition instantanée vers l'état-gare « $\bullet F$ » et une transition étiquetée par F vers l'état « $E \rightarrow \dots F \bullet \dots$ »

Exemple

$$\begin{aligned} S &::= E \# \\ E &::= E - T \mid T \\ T &::= n \mid (E) \end{aligned}$$


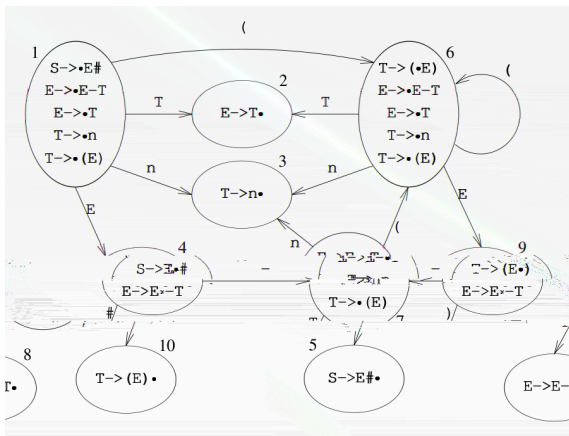
Détermination de l'automate

- | On peut appliquer la méthode de détermination des automates finis.
- | L'automate obtenu s'appelle l'automate LR(0).

Exercice

Déterminez l'automate précédent.

Exemple : Automate LR(0)



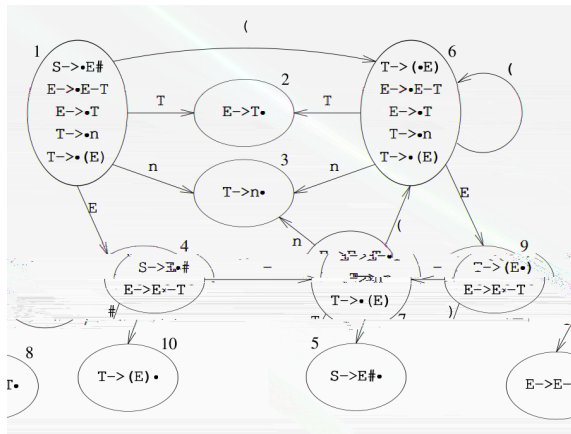
tirée de "Parsing Techniques - A Practical Guide" – Grune, Jacobs

Utilisation de l'automate LR(0)

- | L'automate LR(0) sert à déterminer la **prochaine réduction** à effectuer.
- | On lit la phrase intermédiaire d'analyse de gauche à droite.
- | Cette phrase intermédiaire correspond à la pile de l'automate.
- | Un état d'acceptation de l'automate doit contenir un unique état d'analyse à réduire (c'est-à-dire de la forme « $N \rightarrow \dots \bullet$ »).
- | On réduit en utilisant la règle indiquée par cet état en réécrivant la phrase intermédiaire.
- | Le procédé s'arrête si :
 - ▶ Acceptation de l'entrée : on a réduit le non-terminal d'entrée de la grammaire.
 - ▶ Rejet de l'entrée : une forme intermédiaire est rejetée par l'automate LR(0).

Exemple d'analyse (naïve) LR(0)

- 1 L'analyse de « n - n - n » par l'automate précédent suit les étapes suivantes :



1. n - n - n #
2. T - n - n #
3. E - n - n #
4. E - T - n #
5. E - n #
6. E - T #
7. E #
8. S

tirée de "Parsing Techniques - A Practical Guide" – Grune, Jacobs

Utilisation efficace de l'automate LR(0)

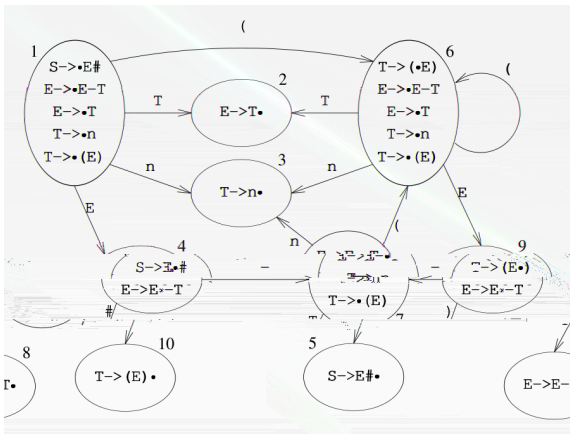
- | Plutôt que de reprendre l'analyse à partir du début de la phrase intermédiaire, on peut se contenter de repartir des derniers états de l'analyse.
- | Pour se souvenir de ces états, on les empile.
- | La pile est donc une succession d'états et de symboles reconnus.
- | L'état courant de l'automate est l'état au sommet de la pile.
- | Nous allons écrire une configuration d'analyse ainsi :

$$s_0 \ S_0 \ s_1 \ S_1 \ \cdots \ s_n \vdash c_1 \ c_2 \ \cdots \ c_n$$

(Le sommet de la pile est à droite.)

Exemple d'analyse LR(0)

1 L'analyse de « n - n - n » par l'automate précédent suit les étapes suivantes :



1. $1 \vdash n - n - n \#$
2. $1 n 3 \vdash - n - n \#$
3. $1 T 2 \vdash - n - n \#$
4. $1 E 4 - 7 n 3 \vdash - n \#$
5. $1 E 4 - 7 T 8 \vdash - n \#$
6. $1 E 4 - 7 n 3 \vdash \#$
7. $1 E 4 - 7 T 8 \vdash \#$
8. $1 E 4 \# 5 \vdash$
9. $S \vdash$

tirée de "Parsing Techniques - A Practical Guide" – Grune, Jacobs

Algorithme LR(0)

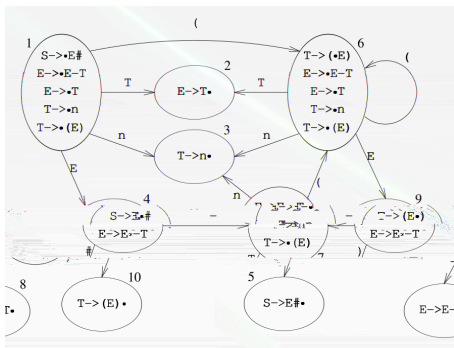
- | L'algorithme d'utilisation de l'automate LR(0) avec cette pile contenant des états et symboles s'énonce ainsi :

1. Soit s , l'état au sommet de la pile.
 - Si s est un état final qui contient une règle de la forme « $N \rightarrow w$ » alors dépiler $|w|$ symboles (et leurs états associés) et réduire (pousser N sur la pile).
 - Si s n'est pas un état final, décaler.
2. Soit S , le symbole au sommet de la pile.
 - Pousser l'état de destination de la transition correspondant au symbole S dans l'automate LR(0), si elle existe.
 - Sinon, rejeter.

Représentation sous forme de tables

- | On représente usuellement l'automate LR(0) sous forme de deux tables.
- | La table ACTION a une colonne et une ligne par état de l'automate.
- | Chaque ligne indique ce qui doit être fait en fonction de l'état au sommet de la pile, c'est-à-dire :
 - ▶ « Décaler » ;
 - ▶ ou bien « Réduire » une règle de la forme « $N \rightarrow w$ ».
- | La table GOTO contient une colonne par symbole de la grammaire et une ligne par état de l'automate.
- | Chaque ligne indique les transitions sortantes de l'état associé.

Exemple d'analyse LR(0)



tirée de "Parsing Techniques - A Practical Guide" – Grune, Jacobs

	ACTION	GOTO						
		n	-	()	#	E	T
1	décale	3	⊥	6	⊥	⊥	4	2
2	$E \rightarrow T$							
3	$T \rightarrow n$							
4	décale	⊥	7	⊥	⊥	5		
5	$S \rightarrow E \#$							
6	décale	3	⊥	6	⊥	⊥	9	2
7	décale	3	⊥	6	⊥	⊥		8
8	$E \rightarrow E - T$							
9	décale	⊥	7	⊥	10	⊥		
10	$E \rightarrow (E)$							

Conflits LR(0)

- | On peut construire un automate LR(0) pour toute grammaire hors-contexte.
- | Malheureusement, l'automate LR(0) n'est pas toujours utilisable.
- | En effet, il peut contenir des conflits :

décalage/réduction ou réduction/réduction

⇒ Une grammaire dont l'automate LR(0) n'a pas de conflits est dite LR(0).

Conflits LR(0)

- | Un conflit « décalage/réduction » apparaît lorsqu'un état de l'automate LR(0) contient un état d'analyse à réduire mais n'est pas un état final de l'automate.
- ⇒ On a le choix entre "décaler T" où T est l'étiquette d'une transition sortante ou bien "réduire" l'état d'analyse à réduire.
- | Un conflit « réduction/réduction » apparaît lorsqu'un état final de l'automate LR(0) contient deux états d'analyse à réduire distincts.
- ⇒ On ne sait pas quelle réduction effectuer.

Algorithme de construction directe de l'automate LR(0)

- | Il n'est pas nécessaire de construire explicitement un automate non-déterministe LR.
- | Un automate déterministe LR peut être directement construit à partir de la grammaire.
- | Il s'agit d'effectuer la déterminisation au fur et à mesure de l'analyse de la grammaire.

Opération auxiliaire

- | On définit l'opération d'**expansion** d'un ensemble d'états d'analyse et d'un états-gare.
- | Pour chaque état, si le \bullet se trouve devant un non-terminal A , on rajoute à l'ensemble des états d'analyse l'ensemble des états d'analyse correspondant aux règles de A .
- | On applique ce processus jusqu'à l'obtention d'un point fixe.
- | Il s'agit de la **fermeture transitive** d'un état d'analyse à travers les transitions instantanées.

Algorithme de construction directe de l'automate LR(0)

- | Soit \mathcal{A} , l'automate LR(0) en cours de construction. On le représente par :
 - ▶ une liste S de paires formées d'un entier et d'un ensemble contenant des états d'analyse et des états-gare. Les entiers servent à numéroter les états de l'automate.
 - ▶ une liste T de transitions « $i \xrightarrow{s} j$ » où i et j sont des numéros d'états et « s » est un symbole (terminal ou non-terminal).
- | Soit \mathcal{L} , une liste d'entiers représentant les états restant à traiter.
- | L'algorithme extrait un élément u de \mathcal{L} , et pour tous les symboles s de la grammaire, fait la chose suivante :

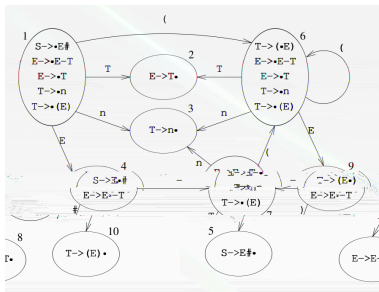
1. Soit \mathcal{E} , un ensemble initialement vide.
2. Pour tous les états d'analyse de l'état u de la forme « $A \rightarrow \bullet s$ », on insère l'état d'analyse « $A \rightarrow s \bullet$ » dans \mathcal{E} .
3. On applique l'opération d'expansion sur \mathcal{E} .
4. Si $\mathcal{E} \notin S$ alors on lui affecte un numéro non utilisé k . On insère (k, \mathcal{E}) dans S , la transition $u \xrightarrow{s} k$ dans T et k dans \mathcal{L} .

Exemple

$$\begin{aligned} S &::= E \# \\ E &::= E - T \mid T \\ T &::= n \mid (E) \end{aligned}$$

Exercice

Appliquez cet algorithme pour reconstruire l'automate :



tirée de "Parsing Techniques - A Practical Guide" – Grune, Jacobs

Les limites des grammaires LR(0)

- | Les grammaires contenant au moins une règle de production vide ne sont pas LR(0).
- | Outre les difficultés posées par les règles de production vide, il s'avère que très peu de grammaire sont LR(0) !
- | Les grammaires LR(0) reconnaissent les langages pour lesquels l'état d'analyse courant suffit à décider ce que l'on doit faire, sans avoir à observer l'entrée.
- | L'entrée sert seulement à confirmer ce choix *a posteriori*

Exemple : une *légère* modification ?

$$\begin{array}{lcl} S & ::= & E \\ E & ::= & E - T \mid T \\ T & ::= & n \mid (E) \end{array}$$

Exercice

Calculez l'automate LR(0) de cette grammaire.

Exemple : une *légère* modification ?

$$\begin{array}{lcl} S & ::= & E \\ E & ::= & E - T \mid T \\ T & ::= & n \mid (E) \end{array}$$

- | L'automate LR(0) contient un conflit décalage/réduction dans l'état contenant les états d'analyse :

$$S \rightarrow E \bullet - T$$

et

$$S \rightarrow E \bullet$$

⇒ En observant le prochain terminal de l'entrée, ce conflit se résout aisément !

Algorithme LR(1)

Une idée déjà rencontrée ...

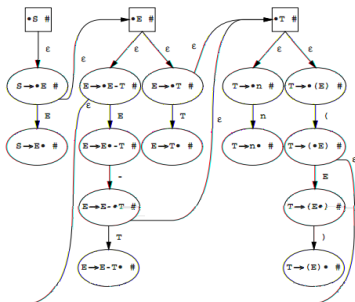
- | Comme pour l'algorithme de Earley, on étend la syntaxe des états d'analyse :

$$N \rightarrow s_1 \dots s_k \bullet s'_1 \dots s'_n \quad [a_1 \dots a_m]$$

Les a_i sont les terminaux attendus en tête de l'entrée une fois que cette analyse sera terminée.

- ⇒ On peut construire un automate fini non déterministe en réutilisant la même méthode que pour l'automate fini non déterministe LR(0). Il suffit seulement en sus de **propager** les terminaux attendus.
- | Comme ces terminaux sont pris en compte pour caractériser les états, l'automate obtenu est par contre beaucoup plus gros !

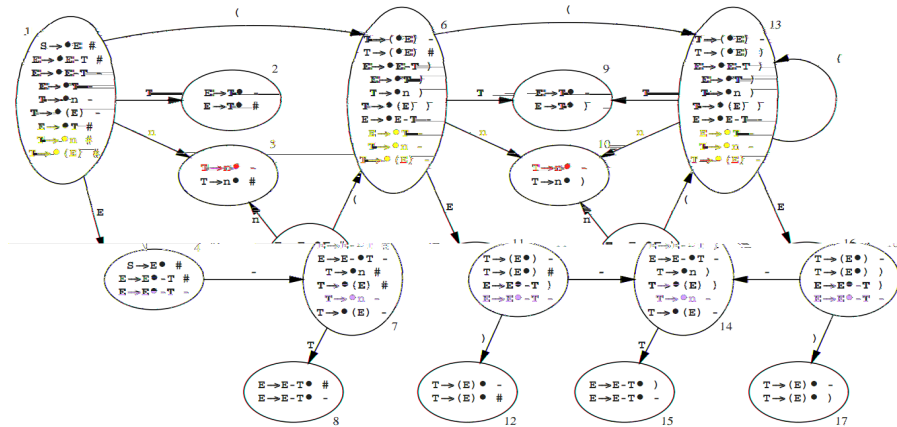
Exemple tirée de "Parsing Techniques - A Practical Guide" – Grune, Jacobs



Propagation des terminaux attendus

- | Les terminaux attendus sont propagés de deux façons distinctes.
- | Des états-gare vers leurs états d'analyses associés : on copie les terminaux attendus.
- | Des états de prédictions de la forme « $A \rightarrow \dots \bullet B s \dots$ » vers l'état-gare concerné (ici, celui de B) :
 - ▶ Si s est un terminal alors c'est le terminal attendu dans la station « $B [s]$ ».
 - ▶ Si s est un non-terminal alors $FIRST(s)$ est l'ensemble des terminaux attendus pour B .

Détermination



Sous forme de tables

	ACTION				
	n	-	()	#
1	d	⊥	d	⊥	⊥
2	⊥	r3	⊥	⊥	r3
3	⊥	r4	⊥	⊥	r4
4	⊥	d	⊥	⊥	r1
6	d	⊥	d	⊥	⊥
7	d	⊥	d	⊥	⊥
8	⊥	r2	⊥	⊥	r2
9	⊥	r3	⊥	r3	⊥
10	⊥	r4	⊥	r4	⊥
11	⊥	d	⊥	d	⊥
12	⊥	r5	⊥	⊥	r5
13	d	⊥	d	⊥	⊥
14	d	⊥	d	⊥	⊥
15	⊥	r2	⊥	r2	⊥
16	⊥	d	⊥	d	⊥
17	⊥	r5	⊥	r5	⊥

	GOTO							
	n	-	()	#	S	E	T
1	3		6			T	4	2
2								
3								
4		7						
6	10		13				11	9
7	3		6					8
8								
9								
10								
11		14		12				
12								
13	10		13				16	9
14	10		13					15
15								
16		14		17				
17								

- | « ⊥ » signifie « rejeter ».
- | « T » signifie « accepter ».
- | « d » signifie « décaler ».
- | « rN » signifie « réduire par la règle N ».

L'automate LR(1)

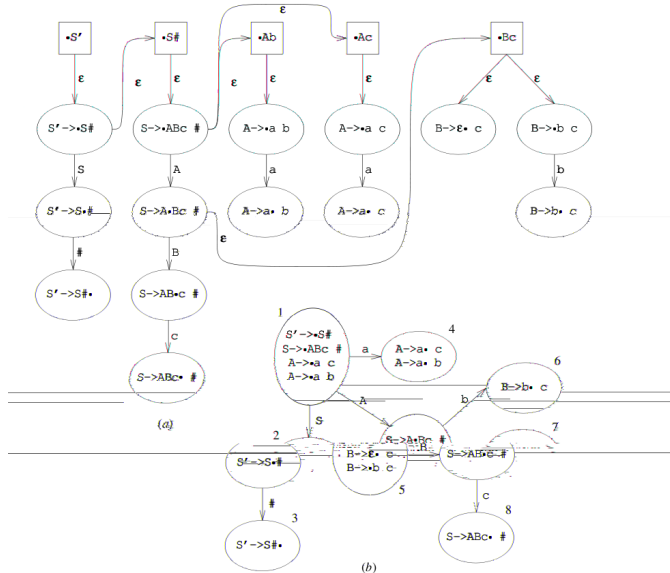
- | L'automate déterministe obtenu est l'automate LR(1).
- | Une grammaire dont l'automate LR(1) n'a pas de conflit est dite LR(1).
- | Deux avantages :
 - ▶ beaucoup de grammaires utiles en pratique sont LR(1) ;
 - ▶ les règles de production vide sont traitées “naturellement” (cf. la suite)
- | Un défaut de taille : le nombre d'états de l'automate est très important !

Les règles de production vide

$$\begin{aligned} S &::= ABc \\ A &::= a \\ B &::= b \mid \epsilon \end{aligned}$$

- | $FIRST(B) = \{b, \epsilon\}$ donc il faut ajouter c aux terminaux attendus après A .
- | Il n'y a pas d'autre difficulté !

Les règles de production vide



Algorithme LALR(1)

Agglomération d'états de l'automate LR(1)

- On appelle **noyau** d'un état de l'automate LR(1), l'ensemble des états d'analyse de cet automate dont on a **e acé**, pour chacun, l'ensemble des terminaux attendus.
- Ainsi l'état 2 de l'automate LR(1) :

$$\begin{aligned} E &\rightarrow T \bullet [-] \\ E &\rightarrow T \bullet [\#] \end{aligned}$$

... et l'état 9 :

$$\begin{aligned} E &\rightarrow T \bullet [-] \\ E &\rightarrow T \bullet [)] \end{aligned}$$

... sont agglomérables en un unique état :

$$\begin{aligned} E &\rightarrow T \bullet [\#] \\ E &\rightarrow T \bullet [-] \\ E &\rightarrow T \bullet [)] \end{aligned}$$

... qui correspond à l'état 2 de l'automate LR(0) mais avec des terminaux attendus associés aux états d'analyse !

Élimination du conflit initial

- | On peut remarquer que le conflit initial de l'automate LR(0) est éliminé par la présence des terminaux attendus, comme dans le cas de l'automate LR(1).
 - | En outre, l'automate obtenu est beaucoup plus petit.
 - | Cet automate se nomme **LALR(1)**, pour
Look Ahead LR(0) with a look-ahead of 1 token
- ⇒ C'est l'automate produit par yacc et bison.
(mais pas par Menhir)

Construction directe de l'automate

- | En pratique, il serait contre-productif d'utiliser le procédé précédent pour construire l'automate LALR(1) puisqu'on s'est appuyé sur l'automate LR(1) !
- ⇒ On peut éviter de **matérialiser** l'automate LR(1) en opérant l'agglomération des états "au vol" pendant la construction de la table.

Ajustement de l'opération d'expansion

- | On doit étendre l'opération d'expansion pour prendre en compte les terminaux attendus.
- | Soit un état (d'analyse ou gare) de la forme « $\dots \bullet A \ [a_1 \dots a_n]$ », on doit rajouter tous les états d'analyse correspondant aux règles de A en calculant les terminaux attendus pour ces règles, c'est-à-dire :
 - ▶ $FIRST(\beta)$ si $\epsilon \notin FIRST(\beta)$
 - ▶ $FIRST(\beta) \cup [a_1, \dots, a_n]$ si $\epsilon \in FIRST(\beta)$

Algorithme de construction directe de l'automate LALR(1)

1. Soit \mathcal{E} , un ensemble initialement vide.
2. Pour tous les états d'analyse de l'état u de la forme
« $A \rightarrow \bullet s [a_1 \dots a_n]$ », on insère dans \mathcal{E} l'état d'analyse
« $A \rightarrow s \bullet [a_1 \dots a_n]$ ».
3. On applique l'opération d'expansion sur \mathcal{E} .
4. Si il n'existe pas d'état dans S possédant le même noyau que \mathcal{E} alors on lui affecte un numéro non utilisé k . On insère dans (k, \mathcal{E}) dans S , la transition $u \xrightarrow{s} k$ dans T et k dans \mathcal{L} .
5. Si il existe un état \mathcal{U} de S possédant le même noyau que \mathcal{E} alors on fusionne \mathcal{U} et \mathcal{E} en un état \mathcal{W} que l'on rajoute dans la liste \mathcal{L} si il est différent de \mathcal{U} .

Exemple

$$\begin{aligned} S &::= E \\ E &::= E Q T \mid T \\ Q &::= - \mid \epsilon \\ T &::= n \mid (E) \end{aligned}$$

Exercice

Calculez l'automate LALR(1) de cette grammaire à l'aide l'algorithme précédent.

Où a-t-on triché?

- | Par malchance, la fusion des états ayant le même noyau n'est pas une opération **conservative** : on peut introduire des conflits dans l'automate LALR qui n'étaient pas présents dans l'automate LR.
- ⇒ Cette situation est très rare mais, lorsqu'elle arrive, elle est difficile à expliquer.
- ⇒ C'est pour cela que Menhir utilise une compression **conservative** de l'automate LR.

Conflits LALR

- | Si l'automate LALR a un conflit « décalage/réduction », alors ce conflit état déjà présent dans l'automate LR.
- | En effet, si un automate contient un état possédant un état d'analyse à réduire et un état d'analyse à décaler alors cela signifie qu'un état de l'automate LR avait un noyau identique et donc, un conflit lui aussi.
- | Par contre, certains conflits « réduction/réduction » peuvent être présents dans l'automate LALR et absents de l'automate LR. . .

Exemple de conflits LALR

S	$::=$	$E\#$
E	$::=$	$aBc \mid bCc \mid aCd \mid bBd$
B	$::=$	e
C	$::=$	e

- | L'automate LR contient un état « $C \rightarrow e \bullet [c], B \rightarrow e \bullet [d]$ » et un **autre** état de même noyau « $C \rightarrow e \bullet [d], B \rightarrow e \bullet [c]$ »
- \Rightarrow La fusion de ces deux états introduit le conflit.

Synthèse

Analyse ascendante

- | Nous avons découvert l'analyse syntaxique LR, c'est-à-dire une classe d'algorithmes s'appuyant sur des automates déterministes pour réaliser une analyse ascendante.
- | Ces automates sont les plus couramment utilisés par les générateurs d'analyseurs syntaxiques.
- | La prochaine fois, nous concluerons ce cours sur l'analyse syntaxique en répondant aux questions suivantes :
 - ▶ Comment construire un arbre de production à l'aide des algorithmes étudiés ?
 - ▶ Comment utiliser Menhir ?
 - ▶ Comment résoudre un conflit ?

INTRODUCTION À LA COMPILATION

Cours 6 : L'analyse syntaxique en pratique

Yann Régis-Gianas
yrg@pps.univ-paris-diderot.fr

PPS - Université Denis Diderot – Paris 7

Résolution des conflits

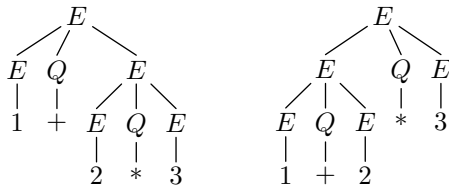
Les conflits de l'analyse syntaxique

- | Une grammaire G est ambiguë si un mot a plusieurs dérivations à travers G .
- | Lorsqu'un mot peut être analysé de plusieurs façons, cela signifie que l'analyse syntaxique mène à plusieurs arbres de production, c'est-à-dire une forêt.
- | Il y a deux solutions à ce problème :
 1. On peut appliquer des critères externes à la grammaire pour filtrer *a posteriori* ces arbres de production, en espérant qu'un unique arbre résulte de ce processus.
 2. On peut appliquer un filtrage au fur et à mesure de l'analyse de façon à ne produire qu'un arbre, au plus.

Exemple

$$\begin{aligned} E &::= E Q E \mid 0 \mid 1 \mid \dots \\ Q &::= + \mid * \end{aligned}$$

| Le mot « $1 + 2 * 3$ » est associé aux arbres de production :



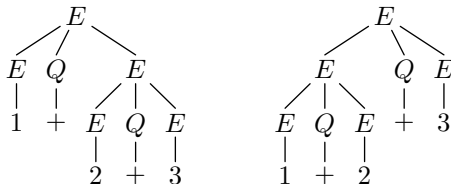
⇒ Le critère est :

La règle « $Q \rightarrow +$ » doit être “au dessus de” la règle « $Q \rightarrow *$ » dans l’arbre.

Exemple

$$\begin{aligned} E &::= E Q E \mid 0 \mid 1 \mid \dots \\ Q &::= + \mid * \end{aligned}$$

Le mot « $1 + 2 + 3$ » est associé aux arbres de production :



⇒ Le critère est :

Une séquence d'application de la règle « $Q \rightarrow +$ » doit former un arbre "qui penche à gauche".

Inconvénients de la résolution *a posteriori*

- | L'application de critères de filtrage sur une forêt d'arbres nécessite un algorithme qui produit effectivement une forêt (ce qui n'est pas le cas des algorithmes déterministes LL et LR que nous avons étudiés).
 - | Certaines ambiguïtés des grammaires peuvent conduire à la construction de forêts de taille exponentielle.
- ⇒ Cela pose des problèmes de représentation en mémoire et le temps nécessaire au filtrage peut s'avérer très important.

Résolution des conflits pour les algorithmes déterministes

- | Les grammaires possédant des conflits LL et LR sont par définition inutilisables avec ces algorithmes.
- | On peut cependant supprimer ces conflits par deux procédés :
 - ▶ l'utilisation d'une information externe à la grammaire (priorité, associativité) ;
 - ▶ la réécriture de la grammaire en une grammaire équivalente.

Conflits LL

- | Nous avons vu qu'il existe 3 types de conflits LL :
 - ▶ FIRST/FIRST
 - ▶ FIRST/FOLLOW
 - ▶ FOLLOW/FOLLOW

Exemple

$$\begin{aligned} E &::= E L E \mid 0 \mid 1 \mid \dots \\ L &::= L Q \mid \epsilon \\ Q &::= T O \mid T \\ O &::= + \mid * \mid \epsilon \\ T &::= . \mid \epsilon \end{aligned}$$

Exercice

Calculer la table de prédiction $LL(1)$ de cette grammaire et classer ses conflits.

Techniques de résolution de conflits LL(1)

- | La récursion à gauche provoque systématiquement un conflit FIRST/FIRST.
- ⇒ Nous savons comment l'éliminer !
 - | Une source de conflits FIRST/FIRST provient du partage d'un préfixe entre plusieurs règles d'un même terminal.
- ⇒ Une **factorisation à gauche** permet de retarder le choix d'une de ces règles.
 - | Parfois, d'autres transformations moins systématiques suffisent mais il faut s'assurer qu'on préserve le langage représenté par la grammaire. . .

Exemple de factorisation à gauche

- La grammaire suivante a un conflit FIRST/FIRST :

$$N ::= ab \mid ac$$

qui est résolu immédiatement par factorisation à gauche :

$$\begin{array}{l} N ::= aM \\ M ::= b \mid c \end{array}$$

Exemple

$$\begin{aligned} E &::= ELE \mid 0 \mid 1 \mid \dots \\ L &::= LQ \mid \epsilon \\ Q &::= TO \mid T \\ O &::= + \mid * \mid \epsilon \\ T &::= . \mid \epsilon \end{aligned}$$

Exercice

Conflits LR(1)

- | Les conflits LR sont de deux natures :
décalage/réduction ou réduction/réduction

Exemple

S	$::=$	$E\#$
E	$::=$	$E \text{ then } E$
	$ $	$0 \mid 1 \mid \dots$
	$ $	$E E$

Exercice

Calculer l'automate LALR(1) de cette grammaire.

Explication du premier conflit

- ** Conflict (shift/reduce) in state 6.
- ** Tokens involved: THEN INT
- ** The following explanations concentrate on token THEN.
- ** This state is reached from s after reading:

e e

- ** The derivations that appear below have the following common factor:

S
e EOF
(?)

- ** In state 6, looking ahead at THEN, reducing production
- ** $e \rightarrow e e$
- ** is permitted because of the following sub-derivation:

e THEN e // lookahead token appears
e e .

- ** In state 6, looking ahead at THEN, shifting is permitted
- ** because of the following sub-derivation:

e e
e . THEN e

Résolution 1 du premier conflit

- | Supposons que l'on désire lire le mot « INT INT THEN INT » ainsi : « INT (INT THEN INT) ».
- | Dans l'automate, cela signifie que l'on donne la priorité dans l'état 6 à l'action de décalage qui passe à l'état 4 à la lecture d'un terminal THEN vis-à-vis de l'action de réduction par la règle 1 « $e \rightarrow e e$ ».
- | Ce choix est exprimé à l'aide d'une priorité : il suffit d'écrire dans le prélude de la spécification de la grammaire que THEN a une priorité plus forte que la règle 1.
- | La priorité d'une règle est donnée par la priorité de son terminal le plus à droite.
- | Or, la règle 1 n'a pas de terminal !
- | On utilise la syntaxe :

`e : e e %prec app`

qui introduit un terminal virtuel « app » qui sert juste à dénoter la priorité de la règle.

Résolution 2 du premier conflit

- Une autre façon de supprimer cet ambiguïté s'obtient en réécrivant la grammaire ainsi :

$$\begin{array}{lcl} S & ::= & E_0 \# \\ E_0 & ::= & E_0 E_1 \\ & | & E_1 \\ E_1 & ::= & 0 \mid 1 \mid \dots \\ & | & E_0 \textbf{ then } E_0 \end{array}$$

- Cette **stratification** de la grammaire explicite les priorités relatives des opérateurs.

Explication du second conflit

- ** Conflict (shift/reduce) in state 5.
- ** Tokens involved: THEN INT
- ** The following explanations concentrate on token THEN.
- ** This state is reached from s after reading:

e THEN e

- ** The derivations that appear below have the following common factor:

S
e EOF
(?)

- ** In state 5, looking ahead at THEN, shifting is permitted
- ** because of the following sub-derivation:

e THEN e
e . THEN e

- ** In state 5, looking ahead at THEN, reducing production
- ** $e \rightarrow e \text{ THEN } e$
- ** is permitted because of the following sub-derivation:

e THEN e // lookahead token appears
e THEN e .

Résolution 1 du second conflit

- | Ce conflit porte sur un décalage et une réduction provoqués par un terminal qui est le non-terminal le plus à droite de la règle à réduire.
- | Supposons que l'on désire lire « INT THEN INT THEN INT »
ainsi : « (INT THEN INT) THEN INT »
- | Cela signifie que l'on veut favoriser la réduction devant le décalage.
- | Il suffit de rajouter une directive « %left THEN » dans le prélude pour exprimer ce choix.
- | La directive « %right THEN » donnerait la priorité au décalage.
- | Tandis que la directive « %nonassoc THEN » rejetterait cette entrée.

Résolution 2 du second conflit

- Encore une 2 fois, on peut réécrire la grammaire pour résoudre ce conflit.

S	$::=$	$E_0 \#$
E_0	$::=$	$E_0 E_1$
	$ $	E_1
E_1	$::=$	$0 \mid 1 \mid \dots$
	$ $	$E_0 \text{ then } E_1$

Exemple de conflit réduction/réduction

S	$::=$	$E \#$
E	$::=$	$E Q_1 E$
	$ $	$E Q_2 E$
	$ $	$0 \mid 1 \mid \dots$
Q_1	$::=$	$+ \mid \epsilon$
Q_2	$::=$	$* \mid \epsilon$

Exercice

Calculer l'automate LALR(1) de cette grammaire.

Résolution des conflits réduction/réduction

- | En général, la résolution des conflits réduction/réduction nécessitent une reformulation de la grammaire.
- | La “bonne” formulation de la grammaire précédente est :

$$\begin{array}{lcl} S & ::= & E \# \\ E & ::= & T E \\ & & | T \\ T & ::= & F + T \\ & & | F \\ F & ::= & n * F \\ & & | 0 \mid 1 \mid \dots \end{array}$$

Une grammaire LR(2)

$$\begin{array}{lcl} G & ::= & R G \mid R \\ R & ::= & 'i' ' : ' D \\ D & ::= & P \\ & & \mid D ' ' P \\ & & \mid D ' ; ' \\ P & ::= & P ' i' \\ & & \mid \epsilon \end{array}$$

- | Après avoir lu « i : D | P », si l'analyseur syntaxique voit un terminal « i », il ne peut pas savoir si ce « i » doit être intégré dans la liste des « i » de P ou bien si ce « i » commence un nouveau R.
 - | Pourtant, en lisant un terminal plus loin, il pourrait décider :
 - ▶ si ce terminal est un ' : ' alors on commence un R ;
 - ▶ si ce terminal est un ' i ' alors on doit continuer l'analyse du P.
- ⇒ Ceci est caractéristique d'une grammaire LR(2).

Synthèse générale sur l'analyse syntaxique

Savoir et savoir-faire

- | À l'issue de cette première partie du cours de compilation sur l'analyse syntaxique, vous devez être en mesure de :
 - ▶ spécifier une grammaire ;
 - ▶ appliquer un algorithme d'analyse syntaxique général (Earley, Unger...) ;
 - ▶ déterminer si une grammaire est $LL(1)$, si elle est $LR(1)$, ... ;
 - ▶ calculer les tables associées à ces algorithmes ;
 - ▶ résoudre les conflits LL ou LR ;
 - ▶ utiliser `MENHIR` pour produire l'analyseur syntaxique d'une grammaire $LR(1)$.

Les générateurs d'analyseurs syntaxiques

- | Dans ce cours, nous utilisons `MENHIR`.
- | Pour chaque langage de programmation, on trouve un outil similaire :
 - ▶ `GNU BISON` : C, C++, ...
 - ▶ `SABLECC` : Java.
 - ▶ `WISENT` : Python.
 - ▶ `HAPPY` : Haskell.
- | Il existe aussi des générateurs d'analyseurs syntaxiques qui s'appuient sur LL :
 - ▶ `ANTLR` : C, C++, C#, Objective-C, Java, Python, ...
 - ▶ `COCO/R` : C, C++, C#, Objective-C, Java, Python, ...

Les algorithmes non vus en cours

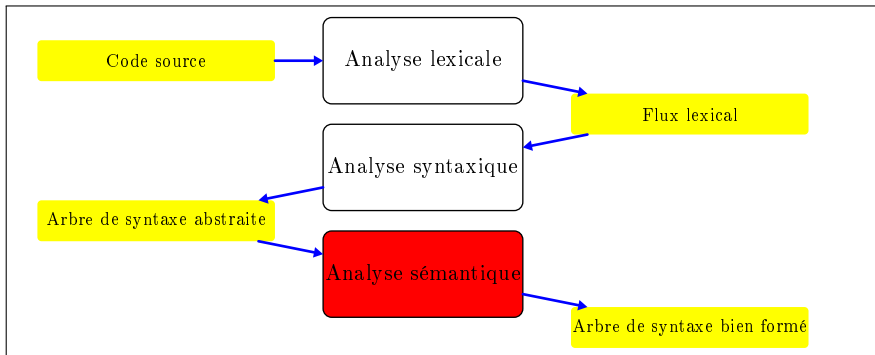
- | Il existe de nombreux raffinements des algorithmes vus en cours.
- | GLR généralise LR en admettant une forme de non-déterminisme qui permet de traiter efficacement l'ensemble des grammaires hors-contextes non ambiguës.
- | Il est implémenté dans les générateurs d'analyseurs syntaxiques :
 - ▶ GNU BISON
 - ▶ ELKHOUND
 - ▶ DYPGEN

INTRODUCTION À LA COMPILATION

Cours 7 : Syntaxe

Yann Régis-Gianas
`yrg@pps.univ-paris-diderot.fr`

PPS - Université Denis Diderot – Paris 7



Qu'appelle-t-on “syntaxe” ?

Préliminaire : vocabulaire et notation

- | À partir de maintenant, nous travaillerons uniquement sur des **arbres**.
- | Plutôt que de dessiner ces arbres, nous continuerons à utiliser une notation textuelle mais qu'il faut interpréter implicitement comme un arbre.
- | On définira des **grammaires d'arbre**, en gardant implicites les priorités et les associativités utilisées pour leur représentation textuelle. Par exemple :

$$\begin{array}{rcl} e & ::= & n \\ & | & e + e \\ & | & e * e \end{array}$$

est une grammaire d'arbres, que nous appellerons aussi abusivement **langage**.

- | Le fait que “1 + 2 * 3” représente l'arbre “1 + (2 * 3)” est implicite.
- | Dans ces grammaires, les non-terminaux sont appelés “méta-variables”.
- | Les arbres obtenus seront appelés **termes** .

Exemple 1 : le langage des expressions arithmétiques

$$\begin{array}{lcl} e & ::= & n \\ & | & e + e \\ & | & e * e \\ & | & e / e \\ & | & e - e \\ & | & -e \end{array}$$

Exemple 2 : le langage des commandes MINILOGO

- | Un programme est une liste d'instructions.
- | Une instruction est de l'une des deux formes suivantes :
 - ▶ Avance N où N est un entier représentant un nombre de pixel.
 - ▶ Tourne N où N est un entier représentant un angle en degré.

devient, plus formellement :

$$\begin{array}{ll} \textit{prog} & ::= \textit{instr}; \textit{prog} \\ & | \textbf{NeFaitRien} \\ \\ \textit{instr} & ::= \textbf{Avance } n \qquad n \in \mathbb{N} \\ & | \textbf{Tourne } d \qquad d \in [0 \dots 359] \end{array}$$

Le sens associé à un terme

- | Les arbres ont une **structure**.
- | Lors du premier cours, nous avons vu qu'il était intuitivement plus facile de donner du sens à un objet structuré plutôt qu'à un objet linéaire.
- | Dans une structure hiérarchique, comme un arbre, on peut *donner un sens à la structure globale en s'appuyant sur le sens de ses sous-structures*.

⇒ Sujet du cours d'aujourd'hui :

La compréhension et la formulation de cette intuition

Comment définir le sens d'un terme ?

- | Le sens d'un terme est une **interprétation** de ce terme.
 - | Avant toute chose, il faut donc se donner un **domaine d'interprétation**.
 - | Il y a une *infinité* de domaines intéressants.
 - | Par exemple, on peut interpréter une expression arithmétique comme :
 - ▶ un calcul dans \mathbb{N} caractérisé par sa valeur finale $v \in \mathbb{N}$;
 - ▶ un calcul dans \mathbb{N} caractérisé par sa valeur finale $v \in \mathbb{N}$ et la séquence des opérations atomiques qui a mené à cette valeur ;
 - ▶ un calcul dans $\mathbb{Z}/n\mathbb{Z}$, c'est-à-dire un calcul sur des entiers bornés ;
 - ▶ un entier représentant la hauteur de son arbre de syntaxe abstraite ;
 - ▶ une formule logique, un entier faisant référence à une variable propositionnelle.
 - ▶ ...
- ⇒ On peut **observer** une interprétation donnée à différents niveaux de détails.
- | La relation associant termes et interprétations est appelée **sémantique**.

À l'aide du langage naturel ?

- | Comment spécifier cette relation ?
- | On ne veut pas énumérer tous les couples (terme, interprétation).
- | La structure d'arbre suggère une définition récursive.

Exemple 1 : le langage des expressions arithmétiques

- | Soit le langage :

$$\begin{array}{lcl} e & ::= & n \\ & | & e + e \\ & | & e * e \\ & | & e / e \\ & | & e - e \\ & | & -e \end{array}$$

- | Une façon naturelle de donner du sens à toute expression e est :
 - ▶ Si $e \equiv n$ alors $\text{sens}(e) = n \in \mathbb{N}$
 - ▶ Si $e \equiv e_1 + e_2$ et $\text{sens}(e_1) = n_1$ et $\text{sens}(e_2) = n_2$ alors $\text{sens}(e) = n_1 +_{\mathbb{N}} n_2$
 - ▶ Si $e \equiv e_1 * e_2$ et $\text{sens}(e_1) = n_1$ et $\text{sens}(e_2) = n_2$ alors $\text{sens}(e) = n_1 *_{\mathbb{N}} n_2$
 - ▶ Si $e \equiv e_1 - e_2$ et $\text{sens}(e_1) = n_1$ et $\text{sens}(e_2) = n_2$ alors $\text{sens}(e) = n_1 -_{\mathbb{N}} n_2$
 - ▶ Si $e \equiv e_1 / e_2$ et $\text{sens}(e_1) = n_1$ et $\text{sens}(e_2) = n_2$ alors $\text{sens}(e) = n_1 /_{\mathbb{N}} n_2$
 - ▶ Si $e \equiv -e_1$ et $\text{sens}(e_1) = n_1$ alors $\text{sens}(e) = -_{\mathbb{N}} n_1$

Exemple 2 : le langage des commandes MINILOGO

$$\begin{array}{lcl} \textit{prog} & ::= & \textit{instr}; \textit{prog} \\ & | & \textbf{NeFaitRien} \end{array}$$
$$\begin{array}{lcl} \textit{instr} & ::= & \textbf{Avance } n \qquad n \in \mathbb{N} \\ & | & \textbf{Tourne } d \qquad d \in [0 \dots 359] \end{array}$$

- Soit S l'ensemble des commandes MINILOGO. Soit $\text{caractérisé}(c)$ la fonction qui associe à une commande c son caractère principal. Soit $\text{position}(c)$ la fonction qui associe à une commande c sa position. Soit $\text{type}(c)$ la fonction qui associe à une commande c son type. Soit $\text{style}(c)$ la fonction qui associe à une commande c son style. Soit $\text{style}(c)$ la fonction qui associe à une commande c son style.

Exemple 3 : appels de procédure

$$\begin{aligned} \text{prog} &::= \text{definition}^*; \text{expression} \\ \text{definition} &::= \mathbf{def} \ f(x_1, \dots, x_n) = \text{expression} \\ \text{expression} &::= \begin{array}{l} n \\ | \\ x \\ | \\ \text{expression} \ \delta \ \text{expression} \\ | \\ f(\text{expression}_1, \dots, \text{expression}_n) \end{array} \\ \delta &::= + \mid - \mid / \mid * \end{aligned}$$

Exercice

Sauriez-vous donner, en français, une sémantique “intéressante” à ce langage ?

À l'aide d'un programme O'CAML ?

```
type binop = Add | Mul | Sub | Div
```

```
type exp = Int of int | Binop of binop × exp × exp
```

```
let rec eval = function
```

```
| Int x → x
```

```
| Binop (Add, e1, e2) → eval e1 + eval e2
```

```
| Binop (Mul, e1, e2) → eval e1 × eval e2
```

```
| Binop (Sub, e1, e2) → eval e1 - eval e2
```

```
| Binop (Div, e1, e2) → eval e1 / eval e2
```

Comment définir mathématiquement une relation sur les termes ?

Propriétés de la fonction O'CAML

```
type binop = Add | Mul | Sub | Div
```

```
type exp = Int of int | Binop of binop  $\times$  exp  $\times$  exp
```

```
let rec eval = function
```

```
| Int x  $\rightarrow$  x  
| Binop (Add, e1, e2)  $\rightarrow$  eval e1 + eval e2  
| Binop (Mul, e1, e2)  $\rightarrow$  eval e1  $\times$  eval e2  
| Binop (Sub, e1, e2)  $\rightarrow$  eval e1 - eval e2  
| Binop (Div, e1, e2)  $\rightarrow$  eval e1 / eval e2
```

- | Cette fonction établit une relation (fonctionnelle) entre un terme et une valeur de type **int**.
- | Elle est définie par récurrence.
- | Que peut-on observer lors de l'évaluation d'une expression ?

Formalisation à l'aide de règles

- On peut s'inspirer de cette fonction pour formaliser l'évaluation à l'aide d'un jugement que l'on écrit :

$$e \Downarrow n$$

que l'on *choisit de lire* :

« L'évaluation de e mène à l'entier n . »

- Le jugement est valide si il est déduit de l'application d'une de ces règles :

(CONST) « $n \Downarrow n$ »

(ADD) « $e_1 + e_2 \Downarrow n_1 +_{\mathbb{N}} n_2$ » si « $e_1 \Downarrow n_1$ » et « $e_2 \Downarrow n_2$ »

(SUB) « $e_1 * e_2 \Downarrow n_1 *_{\mathbb{N}} n_2$ » si « $e_1 \Downarrow n_1$ » et « $e_2 \Downarrow n_2$ »

(MUL) « $e_1 - e_2 \Downarrow n_1 -_{\mathbb{N}} n_2$ » si « $e_1 \Downarrow n_1$ » et « $e_2 \Downarrow n_2$ »

(DIV) « $e_1 / e_2 \Downarrow n_1 /_{\mathbb{N}} n_2$ » si « $e_1 \Downarrow n_1$ » et « $e_2 \Downarrow n_2$ »

- Par exemple, « $1 + 2 + 3 \Downarrow 6$ » car « $1 + 2 \Downarrow 3$ » et « $3 \Downarrow 3$ » ; et « $1 + 2 \Downarrow 3$ » car « $1 \Downarrow 1$ » et « $2 \Downarrow 2$ ».

Arbre de preuve

$$\begin{array}{c} \text{(ADD)} \frac{1 \Downarrow 1 \quad 2 \Downarrow 2}{1 + 2 \Downarrow 3} \quad \frac{}{3 \Downarrow 3} \text{(CONST)} \\ \text{(ADD)} \frac{}{1 + 2 + 3 \Downarrow 6} \end{array}$$

- La dérivation précédente se représente sous la forme d'un arbre.

Règle d'inférence

$$\frac{H_1 \dots H_n}{C}$$

- | Une règle d'inférence est applicable si il existe un arbre de preuve pour chaque H_i , les hypothèses de la règle. On obtient alors un arbre de preuve pour la validité du jugement C , la conclusion de la règle.
- | Une règle sans hypothèse est un axiome.

Les règles de l'évaluation des expressions arithmétiques

$$\frac{}{n \Downarrow n} \quad \text{(CONST)}$$

$$\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2}{e_1 + e_2 \Downarrow n_1 +_{\mathbb{N}} n_2} \quad \text{(ADD)}$$

$$\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2}{e_1 * e_2 \Downarrow n_1 *_{\mathbb{N}} n_2} \quad \text{(MUL)}$$

$$\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2}{e_1 - e_2 \Downarrow n_1 -_{\mathbb{N}} n_2} \quad \text{(SUB)}$$

$$\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2}{e_1 / e_2 \Downarrow n_1 /_{\mathbb{N}} n_2} \quad \text{(DIV)}$$

- | Il faut **instancier** les méta-variables de ces règles pour les utiliser.
- | Quel objet mathématique représente-t-on avec cet ensemble de règles ?

Relation inductive

- | Soit un ensemble \mathcal{U} .
- | Soit un ensemble de règles $(R_i)_{i \in I}$.
- | On définit la fonctionnelle :

$$\left\{ \begin{array}{ll} \mathfrak{P}(\mathcal{U}) & \rightarrow \mathfrak{P}(\mathcal{U}) \\ A & \mapsto \bigcup_{i \in I} \{ R_i(a_1, \dots, a_n) \mid a_1, \dots, a_n \in A, n \equiv \text{arite}(R_i) \} \end{array} \right.$$

- | Elle possède un plus petit point fixe (théorème de Tarski).

Exemple 1 : « être pair »

$$\frac{}{Pair(0)}$$

$$\frac{Pair(n)}{Pair(n+2)}$$

Exemple 2 : « être un arbre binaire de recherche »

$$\begin{array}{c} \overline{ADR(\bullet)} \quad \frac{ADR(t_1) \quad ADR(t_2) \quad PP(x, t_1) \quad PG(x, t_2)}{ADR(\text{nœud}(t_1, x, t_2))} \\[2ex] \frac{PP(x, t_1) \quad PP(x, t_2) \quad x > y}{PP(x, \text{nœud}(t_1, y, t_2))} \quad \overline{PP(x, \bullet)} \\[2ex] \frac{PG(x, t_1) \quad PG(x, t_2) \quad x < y}{PG(x, \text{nœud}(t_1, y, t_2))} \quad \overline{PG(x, \bullet)} \end{array}$$

- | $\bullet \equiv$ “l'arbre vide”.
- | $ADR(t) \equiv$ “ t est un arbre binaire de recherche”
- | $PP(x, t) \equiv$ “tous les entiers de t sont plus petits que x ”
- | $PG(x, t) \equiv$ “tous les entiers de t sont plus grands que x ”

Quelques définitions inductives ...

Définition inductive des termes

- | On peut définir les termes de façon inductive.
- | Soit Σ , un ensemble et une fonction *arite* : $\Sigma \rightarrow \mathbb{N}$
- | Soit \mathcal{U} , l'ensemble des arbres étiquetés par des éléments de Σ .

$$(\text{CONSTRUCT}) \quad \frac{t_1 \dots t_n}{f(t_1, \dots, t_n)} \quad \text{si } n = \text{arite}(f)$$

- | Cette règle a une **condition de bord**.
- | Elle contraint l'application de la règle.
- | Les seuls arbres “bien formés” sont ceux qui respectent l'arité de f .
- | L'ensemble obtenu s'appelle une Σ -algèbre.

Exemple 1 : Les multiplications

- | On choisit $\Sigma = \{0, 1, \dots, *_2\}$.
- | L'indice 2 signifie que l'opérateur $*$ est d'arité 2.

Exemple 2 : Les expressions arithmétiques

- | On choisit $\Sigma = \{0, 1, \dots, *_2, +_2, -_2, /_2\}$.

Définition d'un prédicat inductif sur les termes

- | Pour raisonner sur les termes nous allons utiliser des prédicats inductifs.
- | Ses prédicats porteront souvent sur les termes.
- | De plus, on aura très souvent une règle par constructeur de la syntaxe.
- | On dit alors que le système de règles obtenu est **dirigé par la syntaxe**.

Exemple : « une multiplication qui contient un 0 »

$$\frac{}{Z(0)} \qquad \frac{Z(e_1)}{Z(e_1 * e_2)} \qquad \frac{Z(e_2)}{Z(e_1 * e_2)}$$

- | $Z(e)$ est lu “l’expression e contient un zéro.”
- | Que dire du jugement $Z(1)$?

Exemple : « une multiplication qui contient un 0 »

$$\frac{}{Z(0)} \qquad \frac{Z(e_1)}{Z(e_1 * e_2)} \qquad \frac{Z(e_2)}{Z(e_1 * e_2)}$$

- | $Z(e)$ est lu “l’expression e contient un zéro.”
 - | Que dire du jugement $Z(1)$?
- ⇒ Il n’est pas **dérivable** à partir de cet ensemble de règles.

Exemple : « une multiplication qui contient un 0 »

$$\frac{}{Z(0)} \qquad \frac{Z(e_1)}{Z(e_1 * e_2)} \qquad \frac{Z(e_2)}{Z(e_1 * e_2)}$$

- | $Z(e)$ est lu “l’expression e contient un zéro.”
 - | Que dire du jugement $Z(1)$?
- ⇒ Il n’est pas **dérivable** à partir de cet ensemble de règles.
- | Ce système est-il dirigé par la syntaxe ?

Exemple : « une multiplication qui contient un 0 »

$$\frac{}{Z(0)} \qquad \frac{Z(e_1)}{Z(e_1 * e_2)} \qquad \frac{Z(e_2)}{Z(e_1 * e_2)}$$

- | $Z(e)$ est lu “l’expression e contient un zéro.”
 - | Que dire du jugement $Z(1)$?
- ⇒ Il n’est pas **dérivable** à partir de cet ensemble de règles.
- | Ce système est-il dirigé par la syntaxe ?
- ⇒ Non ! Il y a deux règles pour le cas “ $e_1 * e_2$ ”

Définition d'une relation inductive sur les termes

- | Les règles que nous avons écrit pour définir l'évaluation d'une expression arithmétique ne définissent pas un prédicat inductif mais une relation inductive entre un terme et un entier.
- | Elles sont dirigées par la syntaxe du terme.
- | De plus, on peut prouver :

$$\forall e \ v_1 \ v_2, e \Downarrow v_1 \wedge e \Downarrow v_2 \Rightarrow v_1 = v_2$$

⇒ Comment ?

Preuve par induction sur les termes

I Par induction sur les termes e :

- ▶ Les cas de base sont les constructeurs d'arité 0. Ici, les entiers. On doit prouver :

$$\forall n, n \Downarrow n \wedge n \Downarrow n \Rightarrow n = n$$

C'est vrai par réflexivité de l'égalité.

- ▶ Soit $e \equiv e_1 + e_2$. L'hypothèse d'induction nous apprend que

$$\begin{aligned} \forall n_1 \ n'_1, e_1 \Downarrow n_1 \wedge e_1 \Downarrow n'_1 &\Rightarrow n_1 = n'_1 \\ \forall n_2 \ n'_2, e_2 \Downarrow n_2 \wedge e_2 \Downarrow n'_2 &\Rightarrow n_2 = n'_2 \end{aligned}$$

Soient n et n' tels que $e_1 + e_2 \Downarrow n$ et $e_1 + e_2 \Downarrow n'$.

Seule règle (Add) peut avoir dérivé chacun de ces jugements.

Pour être dérivé, le premier (resp. le second) jugement a nécessairement utilisé un certain k_1 (resp. k'_1) et une dérivation de $e_1 \Downarrow k_1$ (resp. $e_1 \Downarrow k'_1$) ainsi qu'un certain k_2 (resp. k'_2) et une dérivation de $e_2 \Downarrow k_2$ (resp. $e_2 \Downarrow k'_2$), tels que $n = k_1 + k_2$ et $n' = k'_1 + k'_2$. Par application de l'hypothèse d'induction, $k_1 = k'_1$ et $k_2 = k'_2$ et donc $n = n'$.

- ▶ On pourrait traiter les autres cas de façon similaire.

Principe de preuve par induction (Rappel)

- | Soit E , un ensemble.
- | Soit $<$, un ordre strict bien fondé.
- | Montrer une propriété P sur E par induction sur $<$, c'est :
 1. Montrer que P est vraie pour tous les éléments minimaux, c'est-à-dire :

$$\forall x, x \text{ minimal} \Rightarrow P(x)$$

2. Montrer que P est vraie pour tout élément x de E si on suppose que P est vraie pour les prédécesseurs de x , c'est-à-dire :

$$\forall x, (\forall y, y < x \Rightarrow P(y)) \Rightarrow P(x)$$

Preuve par induction sur une relation ou un prédicat inductif

- | Dans le cas des arbres de preuve, la relation d'ordre « est une hypothèse de » est un ordre strict. On peut donc montrer des propriétés vraies pour toute dérivation d'une certaine relation ou d'un prédicat.

Exemple de preuve par induction sur une relation inductive

- Montrons que :

$$\forall e, Z(e) \Rightarrow \forall n, e \Downarrow n \Rightarrow n = 0$$

par induction sur les arbres de dérivation de la forme $Z(e)$, pour tout e .

- Cas de base. Ici, il s'agit de l'axiome :

$$\overline{Z(0)}$$

Cela signifie que $e \equiv 0$. Or, si on a aussi une dérivation de $0 \Downarrow n$, cela implique nécessairement que $n = 0$.

- Cas d'induction. Ici, il y a deux sous-cas :

$$\frac{Z(e_1)}{Z(e_1 * e_2)}$$

$$\frac{Z(e_2)}{Z(e_1 * e_2)}$$

Supposons être dans le premier cas. On a par induction :

$$Z(e_1) \Rightarrow \forall n, e_1 \Downarrow n \Rightarrow n = 0$$

De plus, la dérivation $e_1 * e_2 \Downarrow n$ résulte nécessairement de l'application de la règle (MUL) donc il existe n_1 tel que $n = n_1 *_{\mathbb{N}} n_2$ et $e_1 \Downarrow n_1$. Or, l'induction nous apprend que $n_1 = 0$ donc $n = 0$.

Comment définir une sémantique ?

Sémantique opérationnelle à grands pas

- | La sémantique que nous avons définie *via* le jugement « $e \Downarrow n$ » est une relation entre un terme e et l'entier n qui résulte de son évaluation.
- | Une telle sémantique est dite à **grands pas** car elle s'attache à l'observation du **résultat** d'un calcul et **non aux étapes précises** qui y mènent.
- | On dit aussi que c'est une sémantique **naturelle**.

Sémantique opérationnelle à petits pas

- | Si on est intéressé par les étapes du calcul, une sémantique à grands pas est plus difficile à utiliser.
- | On utilise couramment un **système de réécriture** pour définir formellement **un pas** d'évaluation.
- | Ce système de réécriture définit un jugement de la forme « $e \rightarrow e'$ » qui se lit : « Une étape d'évaluation réécrit le terme e en le terme e' . »
- | Une sémantique spécifiée ainsi est dite à **petits pas** ou encore **structurelle**.

Exemple

$$\frac{e_1 \rightarrow e'_1}{e_1 \oplus e_2 \rightarrow e'_1 \oplus e_2}$$

$$\frac{e_2 \rightarrow e'_2}{n_1 \oplus e_2 \rightarrow n_1 \oplus e'_2}$$

$$\frac{}{n_1 \oplus n_2 \rightarrow m} \quad \text{avec } m = n_1 \oplus_{\mathbb{N}} n_2$$

|

Classification des règles

- | On note deux types de règles.
- | Les règles de **passage au contexte** :

$$\frac{e_1 \rightarrow e'_1}{e_1 \oplus e_2 \rightarrow e'_1 \oplus e_2} \qquad \frac{e_2 \rightarrow e'_2}{n_1 \oplus e_2 \rightarrow n_1 \oplus e'_2}$$

qui permettent la réécriture d'un sous-terme.

- | et les règles de **réduction** :

$$\overline{n_1 \oplus n_2 \rightarrow m} \quad \text{avec } m = n_1 \oplus_{\mathbb{N}} n_2$$

qui transforme le terme en un terme “plus proche” du résultat final.

Relation entre sémantique à grands pas et à petits pas

- | Soit la fermeture transitive et réflexive “ \rightarrow^* ” de la relation “ \rightarrow ”.
- | Montrons que :

$$\forall e \, n, (e \rightarrow^* n) \Leftrightarrow (e \Downarrow n)$$

Sémantique dénotationnelle

- | Il existe des sémantiques **dénotationnelles** qui interprètent les termes par des objets mathématiques.
- | Typiquement, pour les expressions arithmétiques, on peut interpréter les termes dans \mathbb{N} . Ainsi, l'interprétation $\llbracket e \rrbracket$ se définit ainsi :

$$\begin{aligned}\llbracket n \rrbracket &= n \\ \llbracket e_1 \oplus e_2 \rrbracket &= \llbracket e_1 \rrbracket \oplus_{\mathbb{N}} \llbracket e_2 \rrbracket\end{aligned}$$

- | Dans ce cadre, les propriétés sur les termes découlent alors immédiatement des propriétés mathématiques du domaine d'interprétation.
 - | Cependant, même si le cas des expressions arithmétiques est très simple, quels objets mathématiques seraient de “bonnes” interprétations des programmes Caml ou Java ?
- ⇒ Nous n'utiliserons pas cette forme de sémantique dans ce cours car ils nécessitent des outils mathématiques hors de notre portée.

Application : sémantique pour un langage avec variables

Exemple : expressions arithmétiques avec un **let**

$$\begin{array}{lcl} e & ::= & n \\ & | & x \\ & | & e + e \\ & | & e * e \\ & | & \text{let } x := e \text{ in } e \end{array} \quad \begin{array}{l} (1) \\ \\ \\ (2) \end{array}$$

- | Le langage des expressions arithmétiques avec deux nouvelles constructions :
 - (1) Une variable x qui **fait référence à un résultat intermédiaire du calcul**.
 - (2) Une introduction de variable qui **nomme un résultat intermédiaire**.

Variables libres d'une expression

- | Quel sens donné à l'expression « **let** $x := 21$ **in** $x + x$ » ?
- ⇒ Les références à x peuvent être substituées par 21 dans l'expression « $x + x$ ».
- | Quel sens donné à l'expression « x » ?
- ⇒ Cette **occurrence** de x est **libre** : le contexte sous lequel elle est apparaît ne permet pas de déterminer sa valeur.
- ⇒ En d'autres termes, la variable x n'est pas **liée** dans cette seconde expression alors que dans la première expression, elle était liée par le **let**.
- | On dit aussi qu'une variable liée est **muette**. En effet, on peut la renommer sans changer le sens de l'expression.

Exercice

Sauriez-vous définir le jugement :

« Au moins une occurrence de x est libre dans e »

?

Variables libres d'une expression

$$\frac{}{x \in \text{FV}(x)}$$

$$\frac{x \in \text{FV}(e_1)}{x \in \text{FV}(e_1 \oplus e_2)}$$

$$\frac{x \in \text{FV}(e_2)}{x \in \text{FV}(e_1 \oplus e_2)}$$

$$\frac{x \in \text{FV}(e_1)}{x \in \text{FV}(\text{let } y := e_1 \text{ in } e_2)}$$

$$\frac{x \in \text{FV}(e_2) \quad x \neq y}{x \in \text{FV}(\text{let } y := e_1 \text{ in } e_2)}$$

Calcul des variables libres d'une expression

```
type binop = Add | Mul | Div | Sub
```

```
type variable = string
```

```
type e =
```

```
| Int of int
```

```
| Binop of binop × e × e
```

```
| Var of variable
```

```
| Let of variable × e × e
```

```
module SSet = Set.Make (String)
```

```
let rec fv : e → SSet.t = function
```

```
| Var x → SSet.singleton x
```

```
| Binop (_, e1, e2) → SSet.union (fv e1) (fv e2)
```

```
| Let (x, e1, e2) → SSet.union (fv e1) (SSet.remove x (fv e2))
```

```
| _ → SSet.empty
```

Expression close

- | Une expression e est **close** si il n'existe pas de variable libre dans e .
- | Intuitivement : si une expression e est close, on peut l'évaluer en un entier.

Substitution

- | On veut définir le **mécanisme calculatoire/de réduction** correspondant à la **substitution** d'une variable x par sa valeur calculée par le membre gauche du **let** qui l'introduit.
- | Intuitivement : si l'expression à réduire est de la forme « **let** $x := n$ **in** e », on doit remplacer toutes les occurrences de x par n dans e .

Exercice

Sauriez-vous définir le jugement « e' est e dans lequel x est substituée par n » ?

Substitution

$$\begin{array}{llll} (n)\{x/n'\} & = & n & \\ (x)\{x/n\} & = & n & \\ (x)\{y/n\} & = & y & \text{si } x \neq y \\ (e_1 \oplus e_2)\{x/n\} & = & (e_1)\{x/n\} \oplus (e_2)\{x/n\} & \\ (\text{let } y := e_1 \text{ in } e_2)\{x/n\} & = & \text{let } y := (e_1)\{x/n\} \text{ in } (e_2)\{x/n\} & \end{array}$$

⇒ Est-ce correct ?

| Non ! Contre-exemple :

$$(\text{let } x := x + 1 \text{ in } x * 2)\{x/1\} = \text{let } x := 1 + 1 \text{ in } 1 * 2$$

| Nous aimerions plutôt obtenir :

$$(\text{let } x := x + 1 \text{ in } x * 2)\{x/1\} = \text{let } x := 1 + 1 \text{ in } x * 2$$

Substitution, définition corrigée

$$\begin{aligned}(n)\{x/n'\} &= n \\(x)\{x/n\} &= n \\(y)\{x/n\} &= y && \text{si } x \neq y \\(e_1 \oplus e_2)\{x/n\} &= (e_1)\{x/n\} \oplus (e_2)\{x/n\} \\(\text{let } y := e_1 \text{ in } e_2)\{x/n\} &= \text{let } y := (e_1)\{x/n\} \text{ in } (e_2)\{x/n\} && \text{si } x \neq y \\(\text{let } x := e_1 \text{ in } e_2)\{x/n\} &= \text{let } x := (e_1)\{x/n\} \text{ in } e_2\end{aligned}$$

Exercice

Montrer que : si dans un terme e , tous les **let** introduisent des noms de variables distincts alors, dans ce cas, la première définition de la substitution est équivalente à cette version corrigée.

Calcul de la substitution

let rec subst x n = **function**

| Var y **when** x = y \rightarrow Int n

| Binop (op, e1, e2) \rightarrow Binop (op, subst x n e1, subst x n e2)

| Let (y, e1, e2) **when** x = y \rightarrow Let (y, subst x n e1, e2)

| Let (y, e1, e2) \rightarrow Let (y, subst x n e1, subst x n e2)

| x \rightarrow x

Sémantique à petits pas

$$\frac{e_1 \rightarrow e'_1}{e_1 \oplus e_2 \rightarrow e'_1 \oplus e_2} \qquad \frac{e_2 \rightarrow e'_2}{n_1 \oplus e_2 \rightarrow n_1 \oplus e'_2}$$

$$\frac{}{n_1 \oplus n_2 \rightarrow m} \quad \text{avec } m = n_1 \oplus_{\mathbb{N}} n_2$$

$$\frac{e_1 \rightarrow e'_1}{\text{let } x := e_1 \text{ in } e_2 \rightarrow \text{let } x := e'_1 \text{ in } e_2}$$

$$\frac{}{\text{let } x := n \text{ in } e_2 \rightarrow (e_2)\{x/n\}}$$

- Quelles sont les règles de passage au contexte ? Les règles de réduction ?

Évaluation des expressions closes

- | On montre que :

$$\forall e, e \text{ close} \Rightarrow \exists n, e \rightarrow^* n$$

(Par induction sur les termes.)

Écriture d'un interprète de la sémantique à petits pas

```
let rec evalstep : e → e = function
  (** Impossible cases. *)
  | (Int n as e) → assert (not (isvalue e)); exit 1 (* By precondition. *)
  | (Var x as e) → assert (isclosed e); exit 1 (* By precondition. *)

  (** Cases. *)
  | Binop (op, Int n1, Int n2) → Int (evalbinop op n1 n2)
  | Binop (op, Int n, e) → Binop (op, Int n, evalstep e)
  | Binop (op, e1, e2) → Binop (op, evalstep e1, e2)
  | Let (x, Int n, e) → subst x n e
  | Let (x, e1, e2) → Let (x, evalstep e1, e2)
```

Critique de cet interprète

- | La substitution effectue un parcours en profondeur du terme pour trouver toutes les occurrences de la variable à substituer.
 - | Une substitution a lieu en chaque nœud **let** de l'arbre de syntaxe.
 - | La complexité en pire cas de cet interprète est donc **quadratique**.
- ⇒ Peut-on faire mieux ?

Tentative pour une sémantique à grands pas

$$\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2}{e_1 \oplus e_2 \Downarrow n_1 \oplus_{\mathbb{N}} n_2}$$

$$\frac{}{n \Downarrow n}$$

$$\frac{e_1 \Downarrow n_1 \quad (e_2)\{x/n_1\} \Downarrow n_2}{\text{let } x := e_1 \text{ in } e_2 \Downarrow n_2}$$

$$\frac{}{x \Downarrow ?}$$

- Si le terme e est supposé clos alors la règle des variables n'est pas nécessaire.

Interprète de la sémantique à grands pas (pour les expressions closes)

```
let rec eval : e → int = function
  (** Impossible cases. *)
  | (Var x as e) → assert (is_closed e); exit 1 (* By precondition. *)

  (** Cases. *)
  | Int n → n
  | Binop (op, e1, e2) → eval_binop op (eval e1) (eval e2)
  | Let (x, e1, e2) → eval (subst x (eval e1) e2)
```

- | Ce programme est plus concis et sensiblement plus efficace que l'interprète à petits pas mais la substitution subsiste ... donc la complexité reste $O(n^2)$.

Nouvelle tentative de sémantique à grands pas

$$\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2}{e_1 \oplus e_2 \Downarrow n_1 \oplus_{\mathbb{N}} n_2}$$

$$\frac{}{n \Downarrow n}$$

$$\frac{e_1 \Downarrow n_1 \quad (e_2)\{x/n_1\} \Downarrow n_2}{\text{let } x := e_1 \text{ in } e_2 \Downarrow n_2}$$

$$\frac{}{x \Downarrow ?}$$

- | On peut relâcher la contrainte « e clos » à condition de **se souvenir** de la valeur des variables libres.
- | Ainsi, plutôt que de calculer la substitution $(e_2)\{x/n_1\}$, on continue l'évaluation avec de e_2 en notant “quelque part” que $x = n_1$.
- | Si on rencontre, au cours de l'évaluation de e_2 , une occurrence de x , on renvoie le résultat n_1 .

Environnement (lexical)

- La structure permettant de se souvenir de l'association entre variable et valeur entière est appelée un **environnement**.
- C'est moralement une liste d'associations. On peut se donner la syntaxe :

$$\Gamma ::= \bullet \\ \quad | \quad \Gamma; (x \mapsto n)$$

- On se donne un jugement « $\Gamma(x) = n$ » défini ainsi :

$$\frac{}{(\Gamma; (x \mapsto n))(x) = n} \qquad \frac{\Gamma(x) = n \quad x \neq y}{(\Gamma; (y \mapsto n'))(x) = n}$$

Exercice

Peut-on dériver le jugement « $(\bullet; (x \mapsto 21); (x \mapsto 42))(x) = 21$ » ?

Implémentation du type abstrait des environnements

```
module Env : sig
  type t
  val bind : t → variable → int → t
  val empty : t
  val lookup : t → variable → int
end = struct
  type t = (variable × int) list
  let empty = []
  let bind e x n = (x, n) :: e
  let lookup e x = List.assoc x e
end
```

Sémantique à grands pas et à environnement

$$\frac{\Gamma \vdash e_1 \Downarrow n_1 \quad \Gamma \vdash e_2 \Downarrow n_2}{\Gamma \vdash e_1 \oplus e_2 \Downarrow n_1 \oplus_{\mathbb{N}} n_2}$$

$$\frac{}{\Gamma \vdash n \Downarrow n}$$

$$\frac{\Gamma \vdash e_1 \Downarrow n_1 \quad \Gamma; (x \mapsto n_1) \vdash e_2 \Downarrow n_2}{\Gamma \vdash \text{let } x := e_1 \text{ in } e_2 \Downarrow n_2}$$

$$\frac{\Gamma(x) = n}{\Gamma \vdash x \Downarrow n}$$

Évaluation

- | À quelles conditions sur l'environnement Γ et le terme e existe-t-il un entier n ainsi qu'une dérivation de « $\Gamma \vdash e \Downarrow n$ » ?

⇒ Réponse lors du prochain cours.

Interprète de la sémantique à grands pas

```
let rec eval : Env.t → e → int =  
  fun env → function  
    | Var x → Env.lookup env x  
    | Int n → n  
    | Binop (op, e1, e2) → eval_binop op (eval env e1) (eval env e2)  
    | Let (x, e1, e2) → eval (Env.bind env x (eval env e1)) e2  
  
let eval : e → int = eval Env.empty
```

- | Grâce à l'environnement, le coût de l'évaluation d'un **let** est constant.
 - | Par contre, l'évaluation d'une variable nécessite un parcours de l'environnement, dont la taille est bornée par la hauteur de l'arbre de syntaxe.
- ⇒ C'est mieux mais peut-on aller plus loin ?

Une implémentation rusée des variables

- On modifie la syntaxe des termes :

$$\begin{array}{lcl} e & ::= & n \\ & | & \hat{t} \\ & | & e + e \\ & | & e * e \\ & | & \mathbf{let} \iota := e \mathbf{in} e \end{array} \quad \begin{array}{l} (1) \\ \\ \\ (2) \end{array}$$

- (1) On utilise des indices (des entiers) pour représenter les variables.
- (2) Les **lets** sont numérotés : leur indice correspond **au nombre de lets rencontrés depuis la racine** du terme.

⇒ L'indice d'une variable est l'indice du **let** qui l'a introduite.

- Ce sont des **indices de De Bruijn**.

Exemple

- Le terme de la grammaire initiale :

let $x := (\text{let } y := 21 \text{ in } y) \text{ in let } z := 20 \text{ in } z + x + 1$

s'écrit dans ce nouveau langage :

let 0 := (let 1 := 21 in $\hat{1}$) in let 1 := 20 in $\hat{1} + \hat{0} + 1$

(L'indice qui suit le **let** n'est pas essentiel puisqu'on peut le calculer facilement.)

Exercice

Sauriez-vous écrire une fonction qui traduit un terme du langage initial vers ce nouveau langage ?

Implémentation de l'environnement

- On peut implémenter l'environnement à l'aide d'un tableau (borné ou extensible) qui suit une discipline de pile :

```
module Env : sig
  type t
  val bind : t → variable → int → t
  val empty : t
  val lookup : t → variable → int
end = struct
  type t = int × int array
  let max_env_size = 42
  let empty = (0, Array.create max_env_size 0)
  let bind (top, stack) x n =
    assert (top = x);
    stack.(top) ← n;
    (top + 1, stack)
  let lookup (_, stack) x = stack.(x)
end
```

Interprète de la sémantique à grands pas

```
let rec eval : int → Env.t → e → int =  
  fun depth env → function  
    | Var x →  
      Env.lookup env x  
    | Int n →  
      n  
    | Binop (op, e1, e2) →  
      evalbinop op (eval depth env e1) (eval depth env e2)  
    | Let (e1, e2) →  
      eval (depth + 1) (Env.bind env depth (eval depth env e1)) e2  
  
let eval : e → int = eval 0 Env.empty
```

- | L'accès aux valeurs des variables se fait en temps constant.
- | Cet interprète est plutôt efficace !

(On peut encore l'améliorer : la variable "depth" ne sert à rien. Pourquoi ?)

Synthèse

Synthèse

- | Définition de relations et prédicats inductifs sur les termes d'un langage.
 - | Explicitation des sémantiques à grands pas et à petits pas.
 - | Preuve de propriétés sur ces sémantiques.
 - | Un premier mécanisme : l'accès à une variable.
- ⇒ Comment garantir que tous les accès aux variables seront valides ?
- ⇒ Comment étendre ce mécanisme aux appels de fonctions ?

INTRODUCTION À LA COMPILATION

Cours 8 : Liaison de noms

Yann Régis-Gianas
yrg@pps.univ-paris-diderot.fr

PPS - Université Denis Diderot – Paris 7

De la bonne liaison des noms

Le langage des expressions arithmétiques avec variables

$$\begin{array}{lcl} e & ::= & n \\ & | & x \\ & | & e + e \\ & | & e * e \\ & | & \text{let } x := e \text{ in } e \end{array} \quad \begin{array}{l} (1) \\ \\ \\ (2) \end{array}$$

Sémantique à grands pas et à environnement

$$\frac{\eta \vdash e_1 \Downarrow n_1 \quad \eta \vdash e_2 \Downarrow n_2}{\eta \vdash e_1 \oplus e_2 \Downarrow n_1 \oplus_{\mathbb{N}} n_2} \qquad \frac{}{\eta \vdash n \Downarrow n}$$
$$\frac{\eta \vdash e_1 \Downarrow n_1 \quad \eta; (x \mapsto n_1) \vdash e_2 \Downarrow n_2}{\eta \vdash \text{let } x := e_1 \text{ in } e_2 \Downarrow n_2} \qquad \frac{\eta(x) = n}{\eta \vdash x \Downarrow n}$$

Une question en suspens :

- À quelles conditions sur l'environnement η et le terme e existe-t-il un entier n ainsi qu'une dérivation de « $\eta \vdash e \Downarrow n$ » ?

La bonne liaison statique des noms

- | On veut s'assurer qu'à tout nom de variable, on peut associer un entier.
- | Du point de vue des règles d'évaluation, on veut s'assurer que l'hypothèse de la règle (VAR) :

$$\eta(x) = n$$

est toujours valide.

Exercice

Sauriez-vous définir un prédicat inductif garantissant cette propriété ?

La bonne liaison statique des noms : définition

- On se donne une syntaxe pour les **environnements de nommage** :

$$\Gamma ::= \bullet$$
$$| \Gamma; x$$

- La **bonne liaison statique des noms dans une expression e dans Γ** , notée

$$\Gamma \vdash e$$

et qui se lit :

« Les occurrences des variables libres de e sont bien liées dans Γ . »

La bonne liaison statique des nom : règles

$$\begin{array}{c} \Gamma \vdash n \\ \hline \Gamma \vdash x \end{array} \quad \frac{x \in \Gamma}{\Gamma \vdash x} \quad \frac{\Gamma \vdash e_1 \quad \Gamma \vdash e_2}{\Gamma \vdash e_1 \oplus e_2} \quad \frac{\Gamma \vdash e_1 \quad \Gamma; x \vdash e_2}{\Gamma \vdash \text{let } x := e_1 \text{ in } e_2}$$

Exercice

Sauriez-vous écrire le programme qui **décide** cette propriété ?

Vérification de la bonne formation des expressions

```
let rec wf env = function
```

```
| EVar x → List.mem x env
```

```
| EBinop (_, e1, e2) → wf env e1 ∧ wf env e2
```

```
| EInt _ → true
```

```
| ELet (x, e1, e2) → wf env e1 ∧ wf (x :: env) e2
```

Termes bloqués

- | Un terme est dit **bloqué** quand :
 - ▶ on ne peut pas l'évaluer ;
 - ▶ et il n'est pas une valeur.
- | En d'autres termes, le terme est dans une configuration d'erreur.
- | Exemples :
 - ▶ Le terme « x » sous l'environnement « \bullet ».
 - ▶ Le terme « $x + y$ » sous l'environnement « $\bullet; (x \mapsto 21)$ ».
 - ▶ Un programme qui écrit dans une zone mémoire non allouée.
 - ▶ Un programme qui additionne une chaîne de caractères et un nombre flottant.

Les expressions dont les variables sont bien liées ...

- La bonne liaison des variables garantit qu'une expression n'est pas bloquée :

$$\forall e, \bullet \vdash e \Rightarrow \exists n, e \Downarrow n$$

La preuve se fait, par exemple, à l'aide d'une induction sur les expressions.

Exercice

Est-ce vrai aussi pour la sémantique à petits pas ? Si oui, prouvez-le !

Vérification de la bonne formation des expressions

```
let rec wf env = function
| EVar x → List.mem x env
| EBinop (_, e1, e2) → wf env e1 ∧ wf env e2
| EInt _ → true
| ELet (x, e1, e2) → wf env e1 ∧ wf (x :: env) e2
```

- | Ce programme OCAML détermine **sans l'évaluer** si une expression **pourra s'évaluer sans erreur.**
- | C'est une **analyse statique.**

Une évaluation plus efficace que l'interprétation

L'interprète « e cace »

```
let rec eval : int → Env.t → e → int =  
  fun env → function  
    | Var x → Env.lookup env x  
    | Int n → n  
    | Binop (op, e1, e2) → evalbinop op (eval env e1) (eval env e2)  
    | Let (e1, e2) → eval (Env.bind env (eval env e1)) e2  
  
let eval : e → int = eval Env.empty
```

- Si on regarde de plus près les opérations élémentaires utilisées par l'interprète écrit lors du dernier cours, on peut encore remarquer une source d'inefficacité liée au fait que l'interprète effectue **le parcours d'un arbre**.
- En effet, pour parcourir un arbre, ici en profondeur, il faut maintenir une **pile** dont la hauteur est proportionnelle à la hauteur de l'arbre.
- Dans ce programme OCAML, la pile est **implicite** : ce sont les empilements des appels récursifs sur la pile de OCAML qui la réalisent.

Vers un modèle de calcul plus « efficace »

- | Pour évaluer une séquence d'instructions, un pointeur sur l'instruction courante et une fonction de transition vers l'instruction suivante suffisent.
 - | Ces opérations élémentaires se font en **temps constant**.
- ⇒ Ce modèle de calcul est donc strictement plus efficace que le précédent.

Peut-on **traduire**
tout programme du langage des expressions arithmétiques
en **un programme équivalent** d'un langage de séquences d'instructions ?

Des mécanismes plus élémentaires de calcul

- | Pour calculer le résultat d'une expression arithmétique, il faut :
 - ▶ des opérateurs internalisant les opérations arithmétiques ;
 - ▶ un moyen de se souvenir de la valeur d'un entier.
- | Pour traiter les variables, il faut pouvoir :
 - ▶ les introduire dans l'espace des définitions courantes.
 - ▶ interroger leur valeur.
 - ▶ les exclure de l'espace des définitions courantes si elles ne sont plus définies.

Un langage de séquences d'instructions

```
programme ::= instruction+  
  
instruction ::= remember n    n ∈ ℕ  
                | add  
                | mul  
                | div  
                | sub  
                | getvar i      i ∈ ℕ  
                | define  
                | undefine
```

- | Dans quel environnement d'exécution ces instructions ont-elles un sens ?

Type abstrait des piles

- | L'ensemble des résultats intermédiaires ainsi que l'ensemble des valeurs des variables peuvent chacun être stocké dans une **pile** d'entiers.

(Notez que ces piles servent à stocker des résultats intermédiaires et non à déterminer quelle est l'instruction suivante à évaluer.)

- | On se donne une syntaxe pour les piles :

$\begin{array}{lcl} \zeta & ::= & \varepsilon \\ & & n : \zeta \end{array}$	$\begin{array}{l} \text{la pile vide} \\ n \text{ est au sommet de } \zeta \end{array}$
---	---

Type abstrait des piles : opérations

- | « $EstVide(\zeta)$ » de type « $Pile \rightarrow \mathbb{B}$ »

$$EstVide(\zeta) \Leftrightarrow \zeta = \varepsilon$$

- | « $Observe(i, \zeta)$ » de type « $\mathbb{N} \times Pile \hookrightarrow \mathbb{N}$ »

$$Observe(i, n_1 : \dots : n_i : \zeta) = n_i$$

- | « $Depile(\zeta)$ » de type « $Pile \hookrightarrow Pile$ » :

$$Depile(n : \zeta) = \zeta$$

- | « $Empile(n, \zeta)$ » de type « $\mathbb{N} \times Pile \rightarrow Pile$ » :

$$Empile(n, \zeta) = n : \zeta$$

- | « $Vide$ » de type « $Pile$ » :

$$Pile = \varepsilon$$

(Ici, le symbole \rightarrow signifie que la fonction est totale tandis que \hookrightarrow indique que la fonction est partielle.)

Sémantique à petits pas des instructions

add :
 $(\zeta_v, \zeta_r) \rightarrow (\zeta_v, \text{Empile}(\text{Observe}(0, \zeta_r) + \text{Observe}(1, \zeta_r)), \text{Depile}(\text{Depile}(\zeta_r)))$

remember n :
 $(\zeta_v, \zeta_r) \rightarrow (\zeta_v, \text{Empile}(n, \zeta_r))$

getvar i :
 $(\zeta_v, \zeta_r) \rightarrow (\zeta_v, \text{Empile}(\text{Observe}(i, \zeta_v), \zeta_r))$

define :
 $(\zeta_v, \zeta_r) \rightarrow (\text{Empile}(\text{Observe}(0, \zeta_r), \zeta_v), \text{Depile}(\zeta_r))$

undefine :
 $(\zeta_v, \zeta_r) \rightarrow (\text{Depile}(\zeta_v), \zeta_r)$

(Les règles pour la multiplication, la soustraction et la division sont omises.)

Sémantique d'une séquence d'instructions

- La sémantique à petit pas d'un programme est donnée par la règle :

$$\frac{prog[pc] : (\zeta_v, \zeta_r) \rightarrow (\zeta'_v, \zeta'_r)}{(pc, \zeta_v, \zeta_r, prog) \rightarrow (pc + 1, \zeta'_v, \zeta'_r, prog)}$$

Ici, « $prog[pc]$ » est l'instruction du programme à la position pc .

Interprète en O'CAML

```
let rec evalinstr vm = function
| Remember x →
  { vm with valstack = Stack.push x vm.valstack }

| BinOp op →
  let op =
    match op with
    | Add → ( + ) | Mul → ( × )
    | Div → ( / ) | Sub → ( - )
  in
  let x = Stack.inspect 0 vm.valstack in
  let y = Stack.inspect 0 vm.valstack in
  { vm with valstack = Stack.push (op x y) (Stack.pop (Stack.pop vm.valstack)) }

| GetVar i →
  { vm with valstack = Stack.push (Stack.inspect i vm.varstack) vm.valstack }

| Define →
  { vm with
    varstack = Stack.push (Stack.inspect 0 vm.valstack) vm.varstack ;
    valstack = Stack.pop vm.valstack
  }

| Undefine →
  { vm with varstack = Stack.pop vm.varstack }
```

Termes bloqués

- | Les configurations bloquées de la machine virtuelle apparaissent lorsque les conditions d'applications des opérations sur les piles ne sont pas réunies, c'est-à-dire, lorsqu'une pile ne contient pas assez d'éléments.

Exercice (Difficile)

Sauriez-vous définir un prédicat qui capture un ensemble intéressant de séquences d'instructions ne pouvant pas mener à une configuration bloquée ?

Compilation vers la machine virtuelle

La fonction de traduction

- | On veut définir une fonction de traduction \mathcal{C} telle que $\mathcal{C}(e)$ est une séquence d'instructions qui calcule le même entier que l'expression « e ».
 - | Cependant, l'observation de la machine virtuelle que décrit la sémantique concerne les deux piles ζ_r et ζ_v .
- ⇒ Quel entier de ces piles doit être considéré comme résultat du calcul ?

Spécification d'une fonction de traduction

- | Par convention, nous allons observer le **sommet** de la pile ζ_r .
- | Une fois cette convention fixée, la spécification de la fonction de compilation (qui dépend de cette convention) peut être formulée ainsi :

$$\forall e, \exists n, \bullet \vdash e \Downarrow n \Rightarrow (0, \varepsilon, \varepsilon, \mathcal{C}(e)) \rightarrow^* (|\mathcal{C}(e)| - 1, \varepsilon, n : \varepsilon, \mathcal{C}(e))$$

⇒ Essayons de définir une fonction \mathcal{C} qui a cette spécification :

- ▶ D'abord, sur le langage des expressions arithmétiques sans variables ;
- ▶ Puis, sur le langage avec variables.

La fonction de traduction sur quelques exemples

- | $\mathcal{C}(42) =$

- | $\mathcal{C}(1 + 2) =$

- | $\mathcal{C}(1 + 2 * 3) =$

La fonction de traduction sur quelques exemples

- | $\mathcal{C}(42) = \text{remember } 42$
- | $\mathcal{C}(1 + 2) =$
- | $\mathcal{C}(1 + 2 * 3) =$

La fonction de traduction sur quelques exemples

- | $C(42) = \text{remember } 42$
- | $C(1 + 2) = \text{remember } 1; \text{remember } 2; \text{add}$
- | $C(1 + 2 * 3) =$

La fonction de traduction sur quelques exemples

- | $C(42) = \text{remember } 42$
- | $C(1 + 2) = \text{remember } 1; \text{remember } 2; \text{add}$
- | $C(1 + 2 * 3) = \text{remember } 1; \text{remember } 2; \text{remember } 3; \text{mul}; \text{add}$

Traduction pour les expressions arithmétiques

$$\begin{array}{lll} \mathcal{C}(n) & = & \text{remember } n \qquad n \in \mathbb{N} \\ \mathcal{C}(e_1 + e_2) & = & \mathcal{C}(e_1); \mathcal{C}(e_2); \text{add} \\ \mathcal{C}(e_1 * e_2) & = & \mathcal{C}(e_1); \mathcal{C}(e_2); \text{mul} \\ \mathcal{C}(e_1 / e_2) & = & \mathcal{C}(e_1); \mathcal{C}(e_2); \text{div} \\ \mathcal{C}(e_1 - e_2) & = & \mathcal{C}(e_1); \mathcal{C}(e_2); \text{sub} \end{array}$$

- | Aurait-on pu écrire une autre fonction de compilation ?

Traduction pour les expressions arithmétiques

$\mathcal{C}(n)$	=	remember n	$n \in \mathbb{N}$
$\mathcal{C}(e_1 + e_2)$	=	$\mathcal{C}(e_1); \mathcal{C}(e_2); \mathbf{add}$	
$\mathcal{C}(e_1 * e_2)$	=	$\mathcal{C}(e_1); \mathcal{C}(e_2); \mathbf{mul}$	
$\mathcal{C}(e_1 / e_2)$	=	$\mathcal{C}(e_1); \mathcal{C}(e_2); \mathbf{div}$	
$\mathcal{C}(e_1 - e_2)$	=	$\mathcal{C}(e_1); \mathcal{C}(e_2); \mathbf{sub}$	

- | Aurait-on pu écrire une autre fonction de compilation ?
- | Oui ! Par exemple, la solution suivante est elle aussi correcte :

$$\mathcal{C}(e_1 + e_2) = \mathcal{C}(e_2); \mathcal{C}(e_1); \mathbf{add}$$

La traduction d'une expression arithmétique ne bloque pas

- | La traduction produit une séquence d'instructions qui ne bloque pas.
- | On peut montrer, par induction, que la traduction $\mathcal{C}(e)$ transforme une pile de valeurs de la forme « ζ_v » en une pile de valeurs de la forme « $n : \zeta_v$ ».
- | À chaque fois qu'une instruction d'opération arithmétique est insérée dans le code compilé, elle est précédée de deux blocs non vides d'instructions, issus de la compilation de deux sous-expressions.
- | Comme l'évaluation de chacun de ces blocs d'instructions introduit un entier sur la pile, on a nécessairement deux entiers sur la pile, ce qui permet d'appliquer l'opération binaire.

La fonction de traduction sur quelques exemples

- | $\mathcal{C}(\text{let } x := 42 \text{ in } x) =$
- | $\mathcal{C}(\text{let } x := 42 \text{ in let } y := x + 1 \text{ in } y) =$
- | $\mathcal{C}(\text{let } x := \text{let } y := 42 \text{ in } y \text{ in } x) =$

La fonction de traduction sur quelques exemples

- | $\mathcal{C}(\text{let } x := 42 \text{ in } x) =$
remember 42 ; define ; getvar 0 ; undefine
- | $\mathcal{C}(\text{let } x := 42 \text{ in let } y := x + 1 \text{ in } y) =$
- | $\mathcal{C}(\text{let } x := \text{let } y := 42 \text{ in } y \text{ in } x) =$

La fonction de traduction sur quelques exemples

- | $\mathcal{C}(\text{let } x := 42 \text{ in } x) =$
remember 42 ; define ; getvar 0 ; undefine
- | $\mathcal{C}(\text{let } x := 42 \text{ in let } y := x + 1 \text{ in } y) =$
remember 42 ; define ; getvar 0 ; remember 1 ; add ; define ; getvar 0 ; undefine ; undefine
- | $\mathcal{C}(\text{let } x := \text{let } y := 42 \text{ in } y \text{ in } x) =$

La fonction de traduction sur quelques exemples

- | $\mathcal{C}(\text{let } x := 42 \text{ in } x) =$
`remember 42 ; define ; getvar 0 ; undefine`
- | $\mathcal{C}(\text{let } x := 42 \text{ in let } y := x + 1 \text{ in } y) =$
`remember 42 ; define ; getvar 0 ; remember 1 ; add ; define ; getvar 0 ;
undefine ; undefine`
- | $\mathcal{C}(\text{let } x := \text{let } y := 42 \text{ in } y \text{ in } x) =$
`remember 42 ; define ; getvar 0 ; undefine ; define ; getvar 0 ; undefine`

Traduction pour les expressions avec variables

$$\begin{aligned} \mathcal{C}(\text{let } x := e_1 \text{ in } e_2) &= \mathcal{C}(e_1); \text{define}; \mathcal{C}(e_2); \text{undefine} \\ \mathcal{C}(x) &= ? \end{aligned}$$

- | L'indice de x dépend du nombre de **lets** traversés depuis la racine de l'expression globale.
- ⇒ Comment le retrouver ?

Traduction pour les expressions avec variables

$$\begin{aligned} \mathcal{C}(\Gamma, \text{let } x := e_1 \text{ in } e_2) &= \mathcal{C}(\Gamma, e_1); \text{define}; \mathcal{C}(\Gamma; x, e_2); \text{undefine} \\ \mathcal{C}(\Gamma, x) &= \text{pos}(\Gamma, x) \end{aligned}$$

où

$$\text{pos}((\Gamma; y), x) = \begin{cases} 1 + \text{pos}(\Gamma, x) & \text{si } x \neq y \\ 0 & \text{si } x = y \end{cases}$$

- | Notez que pos est une fonction partielle.

Compilateur en O'CAML

```
let rec compile : variable list  $\rightarrow$  e  $\rightarrow$  instruction list =  
  fun env  $\rightarrow$  function  
    | EVar x  $\rightarrow$   
      [ GetVar (positionof x env) ]  
  
    | EBinop (op, e1, e2)  $\rightarrow$   
      compile env e1 @ compile env e2 @ [ BinOp op ]  
  
    | EInt n  $\rightarrow$   
      [ Remember n ]  
  
    | ELet (x, e1, e2)  $\rightarrow$   
      compile env e1 @ [ Define ] @ compile (x :: env) e2 @ [ Undefine ]
```

La traduction d'un programme bien formé ne bloque pas

- | Une expression dont les variables sont bien liées se compile en une séquence d'instructions qui ne bloque pas.
- | La propriété de bonne formation du programme implique que durant l'exécution du programme compilé, la taille de la pile des variables est toujours supérieur aux indices suivant l'instruction **getvar**.
- | Le théorème à prouver est :

$$\begin{aligned} & \forall e \zeta_v \zeta_r \Gamma, \\ & \Gamma \vdash e \wedge |\zeta_v| = |\Gamma| \\ & \Rightarrow \exists n, (0, \zeta_v, \zeta_r, \mathcal{C}(e)) \rightarrow^* (|\mathcal{C}(e)| - 1, \zeta_v, n : \zeta_r, \mathcal{C}(e)) \end{aligned}$$

- | Elle permet de déduire facilement :

$$\forall e, \bullet \vdash e \Rightarrow \exists n, (0, \varepsilon, \varepsilon, \mathcal{C}(e)) \rightarrow^* (|\mathcal{C}(e)| - 1, \varepsilon, n : \varepsilon, \mathcal{C}(e))$$

La liaison de nom **dynamique**

Différences entre liaison de noms statique et dynamique

- | Dans les langages de programmation à **portée statique**, l'évaluation d'une variable est donnée par la valeur de l'expression à laquelle elle est liée dans **le contexte statique introduit par un *let***.
- | Exemples : C, JAVA, OCAML...
- ⇒ C'est le mécanisme de liaison statique des variables expliqué précédemment.
- | Dans les langages de programmation à **portée dynamique**, l'évaluation d'une variable est donnée par **le contexte dynamique dans lequel elle s'évalue**.
- | Exemples : LISP, ...
- ⇒ Qu'est-ce que cela signifie ?

Exemple 1 : En LISP

```
;; Définit une variable globale "x".
```

```
(defvar *x* 10)
```

```
;; Définit une fonction "foo".
```

```
(defun foo ()
```

```
  (format t "Before assignment~18tX: ~d~%" *x*)
```

```
  (setf *x* (+ 1 *x*)) ; ; Référence à une variable 'x' du le contexte.
```

```
  (format t "After assignment~18tX: ~d~%" *x*))
```

```
;; Définit une fonction "bar".
```

```
(defun bar ()
```

```
  (foo) ; ; Cet appel a che 10 puis 11
```

```
  (let ((*x* 20)) (foo)) ; ; Cet appel a che 20 puis 21
```

```
  (foo)) ; ; Cet appel a che 10 puis 11
```

Connaissez-vous JAVASCRIPT ?

Exemple 1 : En JAVASCRIPT

```
var x = 0;  
if (x == 0) {  
    function f () { return 0; }  
}  
f ();
```

Que fait ce programme ?

Exemple 2 : En JAVASCRIPT

```
var x = 0 ;  
function f() {  
    x = 1 ;  
}  
f () ;  
alert (x) ;
```

Que fait ce programme ?

Exemple 3 : En JAVASCRIPT

```
var x = 0;  
function f() {  
    x = 1;  
    if (0) { var x = 2; }  
}  
f ();  
alert (x);
```

Que fait ce programme ?

Synthèse

Synthèse

- | On peut décider si une expression avec variable peut s'évaluer sans erreur.
- | La compilation du langage des expressions arithmétiques avec variables explicitent l'ordre d'évaluation en linéarisant l'arbre de syntaxe abstraite en une séquence d'instructions effectuant le même calcul.

INTRODUCTION À LA COMPILATION

Cours 9 : Langage du premier ordre

Yann Régis-Gianas
yrg@pps.jussieu.fr

PPS - Université Denis Diderot – Paris 7

Le flot de contrôle

Comment programmer des fonctions non constantes ?

- | Le langage des expressions arithmétiques caractérise des entiers naturels.
- | Or, pour être utile, un programme produit généralement un résultat **en fonction** de ses entrées.

Les booléens et le branchement

- On étend le langage des expressions arithmétiques :

e	$::=$	n	
		x	
		$e + e$	
		$e * e$	
		let $x := e$ in e	
		true false	(0)
		if e then e else e	(1)
		$x \mathcal{R}^? y$	(2)

- (0) est un **booléen**.
- (1) est une **expression conditionnelle**.
- (2) est la décision d'une propriété : ici, nous choisirons $\mathcal{R} \in \{<, >, \leq, \geq, =\}$.
- Le résultat v de l'évaluation d'une expression peut être n , **true** ou **false**.

Sémantique opérationnelle à grands pas

$$\frac{\vdash e_1 \Downarrow n_1 \quad \vdash e_2 \Downarrow n_2}{\vdash e_1 \oplus e_2 \Downarrow n_1 \oplus_{\mathbb{N}} n_2}$$

$$\frac{}{\vdash n \Downarrow n}$$

$$\frac{}{\vdash \text{false} \Downarrow \text{false}}$$

$$\frac{}{\vdash \text{true} \Downarrow \text{true}}$$

$$\frac{(x) = v}{\vdash x \Downarrow v}$$

$$\frac{\vdash e_1 \Downarrow v_1 \quad ; (x \mapsto v_1) \vdash e_2 \Downarrow v_2}{\vdash \text{let } x := e_1 \text{ in } e_2 \Downarrow v_2}$$

$$\frac{\vdash e_c \Downarrow \text{true} \quad \vdash e_1 \Downarrow v_1}{\vdash \text{if } e_c \text{ then } e_1 \text{ else } e_2 \Downarrow v_1}$$

$$\frac{\vdash e_c \Downarrow \text{false} \quad \vdash e_2 \Downarrow v_2}{\vdash \text{if } e_c \text{ then } e_1 \text{ else } e_2 \Downarrow v_2}$$

$$\frac{\vdash e_1 \Downarrow n_1 \quad \vdash e_1 \Downarrow n_2}{\vdash e_1 \mathcal{R}^? e_2 \Downarrow \text{true}} \quad \text{si } n_1 \mathcal{R} n_2 \text{ est vrai}$$

$$\frac{\vdash e_1 \Downarrow n_1 \quad \vdash e_1 \Downarrow n_2}{\vdash e_1 \mathcal{R}^? e_2 \Downarrow \text{false}} \quad \text{si } n_1 \mathcal{R} n_2 \text{ est faux}$$

L'interprète en OCAML : les valeurs

```
type value =  
  | VInt of int  
  | VBool of bool
```

```
exception NotInt
```

```
let as_int = function  
  | VInt x  $\rightarrow$  x  
  | _  $\rightarrow$  raise NotInt
```

```
exception NotBool
```

```
let as_bool = function  
  | VBool b  $\rightarrow$  b  
  | _  $\rightarrow$  raise NotBool
```

L'interprète en OCAML : l'AST

```
type binop = Add | Mul | Div | Sub
```

```
type comparison = Le | Ge | Lt | Gt | Eq
```

```
type variable = string
```

```
type t =
```

```
| Int of int
```

```
| Bool of bool
```

```
| Binop of binop  $\times$  t  $\times$  t
```

```
| Let of variable  $\times$  t  $\times$  t
```

```
| Var of variable
```

```
| If of t  $\times$  t  $\times$  t
```

```
| Dec of comparison  $\times$  t  $\times$  t
```


L'interprète en OCAML

```
let rec eval env = function
| Int i →
  VInt i
| Bool b →
  VBool b
| Binop (op, e1, e2) →
  VInt ((binop op) (as_int (eval env e1)) (as_int (eval env e2)))
| Let (x, e1, e2) →
  eval (Env.bind env x (eval env e1)) e2
| Var x →
  Env.lookup env x
| Dec (c, e1, e2) →
  VBool ((comparison c) (as_int (eval env e1)) (as_int (eval env e2)))
| If (c, e1, e2) →
  if as_bool (eval env c) then eval env e1 else eval env e2
```

Bonne formation des expressions

Exercice

Quelles sont les expressions bloquées vis-à-vis de cette sémantique ?

$$\begin{array}{c} \frac{\vdash e_1 \Downarrow n_1 \quad \vdash e_2 \Downarrow n_2}{\vdash e_1 \oplus e_2 \Downarrow n_1 \oplus_{\mathbb{N}} n_2} \qquad \frac{}{\vdash n \Downarrow n} \qquad \frac{}{\vdash \text{false} \Downarrow \text{false}} \\[10pt] \frac{}{\vdash \text{true} \Downarrow \text{true}} \qquad \frac{(x) = v}{\vdash x \Downarrow v} \qquad \frac{\vdash e_1 \Downarrow v_1 \quad ; (x \mapsto v_1) \vdash e_2 \Downarrow v_2}{\vdash \text{let } x := e_1 \text{ in } e_2 \Downarrow v_2} \\[10pt] \frac{\vdash e_c \Downarrow \text{true} \quad \vdash e_1 \Downarrow v_1}{\vdash \text{if } e_c \text{ then } e_1 \text{ else } e_2 \Downarrow v_1} \qquad \frac{\vdash e_c \Downarrow \text{false} \quad \vdash e_2 \Downarrow v_2}{\vdash \text{if } e_c \text{ then } e_1 \text{ else } e_2 \Downarrow v_2} \\[10pt] \frac{\vdash e_1 \Downarrow n_1 \quad \vdash e_1 \Downarrow n_2}{\vdash e_1 \mathcal{R}^? e_2 \Downarrow \text{true}} \quad \text{si } n_1 \mathcal{R} n_2 \text{ est vrai} \\[10pt] \frac{\vdash e_1 \Downarrow n_1 \quad \vdash e_1 \Downarrow n_2}{\vdash e_1 \mathcal{R}^? e_2 \Downarrow \text{false}} \quad \text{si } n_1 \mathcal{R} n_2 \text{ est faux} \end{array}$$

Caractériser la forme de la valeur d'une expression

1. Certaines expressions sont “clairement” à valeur dans les entiers naturels tandis que d'autres sont à valeur dans l'ensemble $\{\mathbf{true}, \mathbf{false}\}$ des booléens :
 - ▶ $0, 1 + 2, \mathbf{let } x := 42 \mathbf{ in } x + 1, \dots$;
 - ▶ $\mathbf{true}, 0 <^? 1, \mathbf{let } b := 0 <^? 1 \mathbf{ in } b, \dots$
2. Certaines expressions sont “clairement” bloquées :
 - ▶ $0 + \mathbf{true}, \mathbf{if } 42 \mathbf{ then } 1 \mathbf{ else } 0, \dots$
3. Certaines ne sont pas nécessairement bloquées mais il faudrait faire un calcul pour s'en assurer et pour connaître la forme de la valeur calculée :
 - ▶ $\mathbf{if } (0 <^? 1 \mathbf{ then true else } 21) \mathbf{ then false else } 42$

Exercice

Existe-t-il une analyse **statique** permettant :

- | d'accepter uniquement les expressions de type 1 ?
- | d'accepter aussi les programmes de type 2 ?

Règles de typage

$$\begin{array}{c} \Gamma \vdash n : \mathbf{nat} \quad \Gamma \vdash \mathbf{true} : \mathbf{bool} \quad \Gamma \vdash \mathbf{false} : \mathbf{bool} \quad \frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \\[2ex] \frac{\Gamma \vdash e_1 : \mathbf{nat} \quad \Gamma \vdash e_2 : \mathbf{nat}}{\Gamma \vdash e_1 \oplus e_2 : \mathbf{nat}} \quad \frac{\Gamma \vdash e_1 : \mathbf{nat} \quad \Gamma \vdash e_2 : \mathbf{nat}}{\Gamma \vdash e_1 \mathcal{R}^? e_2 : \mathbf{bool}} \\[2ex] \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma; x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{let } x := e_1 \mathbf{ in } e_2 : \tau_2} \\[2ex] \frac{\Gamma \vdash e_c : \mathbf{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathbf{if } e_c \mathbf{ then } e_1 \mathbf{ else } e_2 : \tau} \end{array}$$

- | Notez que le type de x dans la règle **let** peut être **déduit** de celui de e_1 .
- ⇒ C'est une forme très restreinte d'**inférence de type**.

Fonction de typage en OCAML

```
let rec typecheck env = function
| Var x → TyEnv.lookup env x
| Int _ → TInt
| Bool _ → TBool
| If (cond, e1, e2) →
    let tycond = typecheck env cond in
    let ty1 = typecheck env e1 and ty2 = typecheck env e2 in
    if tycond ≠ TBool ∨ ty1 ≠ ty2 then raise IllTyped;
    ty1
| Let (x, e1, e2) →
    let ty1 = typecheck env e1 in
    typecheck (TyEnv.bind env x ty1) e2
| Binop (_, e1, e2) →
    let ty1 = typecheck env e1 and ty2 = typecheck env e2 in
    if (ty1 ≠ TInt ∨ ty2 ≠ TInt) then raise IllTyped;
    TInt
| Dec (_, e1, e2) →
    let ty1 = typecheck env e1 and ty2 = typecheck env e2 in
    if (ty1 ≠ TInt ∨ ty2 ≠ TBool) then raise IllTyped;
    TBool
```

Théorème de correction du typage

- On peut montrer que :

$$\forall e \tau, \bullet \vdash e : \tau \Rightarrow \exists v : \tau, \bullet \vdash e \Downarrow v$$

Compilation vers une machine virtuelle

- | Les programmes de la machine virtuelle définie lors du dernier cours sont des séquences d'instruction évaluées de la première à la dernière.
- | En présence de branchement, on ne veut exécuter qu'un **sous-ensemble** des instructions, suivant le chemin d'exécution caractérisé par l'issue des tests.
- | En d'autres termes, il s'agit de **contrôler quelles opérations sont évaluées**.
- | C'est le rôle des **opérateurs de flot de contrôle**.

Nouveau langage pour la machine virtuelle

programme ::= $(\ell : \text{instruction})^+$

instruction ::= **remember** n $n \in \mathbb{N}$
| **add** | **mul** | **div** | **sub**
| **cmple** | **cmpge** | **cmpeq**
| **cmplt** | **cmpgt**
| **getvar** i $i \in \mathbb{N}$
| **define**
| **undefine**
| **branch** ℓ | **branchif** ℓ, ℓ

- | Les étiquettes ℓ sont des noms symboliques pour les emplacements des instructions dans le programme.
- | Les deux dernières instructions sont des opérateurs de contrôle.

Une machine virtuelle à pile

- | Les opérateurs de flot influencent la fonction de transition du contrôle.
- | On doit maintenant expliciter le flot du contrôle dans la sémantique :

$$instruction : (pc, \zeta_v, \zeta_r) \rightarrow (pc', \zeta'_v, \zeta'_r)$$

et qui se lit :

« L'instruction à la position pc du programme transforme les piles (ζ_v, ζ_r) en (ζ'_v, ζ'_r) et transporte l'exécution à la position pc' . »

Sémantique à petits pas des instructions

op : (Pour toutes les opérations définies dans le dernier cours.)
 $(pc, v, r) \rightarrow (pc + 1, \dots, \dots)$

branch :
 $(pc, v, r) \rightarrow (, v, r)$

branchif $_1, _2$: // Si $Observe(r, 0) \equiv \text{true}$
 $(pc, v, r) \rightarrow (_1, v, Depile(r))$

branchif $_1, _2$: // Si $Observe(r, 0) \equiv \text{false}$
 $(pc, v, r) \rightarrow (_2, v, Depile(r))$

Interprète en OCAML

```
let rec evalinstr labels vm = function
| Branch l →
    goto labels vm l

| BranchIf (l1, l2) →
    let x = Stack.inspect 0 vm.valstack in
    let vm =
        { vm with valstack = Stack.pop vm.valstack }
    in
    (match x with
     | VBool true → goto labels vm l1
     | VBool false → goto labels vm l2
     | _ → raise NotBool)

| ...
and goto labels vm l =
    evalinstr labels vm (snd (vm.prog.(pos_of_label labels l)))
```

La fonction de traduction sur quelques exemples

| $\mathcal{C}(\text{if } 0 \leq 1 \text{ then } 0 \text{ else } 1) =$

La fonction de traduction sur quelques exemples

| $\mathcal{C}(\text{if } 0 \leq 1 \text{ then } 0 \text{ else } 1) =$

```
remember 1
remember 0
cmple
branchif  $\ell_1, \ell_2$ 
 $\ell_1$  : remember 0
      branch  $\ell_3$ 
 $\ell_2$  : remember 1
      branch  $\ell_3$ 
 $\ell_3$  :
```

Traduction

$\mathcal{C}(\text{if } e_c \text{ then } e_1 \text{ else } e_2)$	$=$	$\mathcal{C}(e_c)$
		branchif ℓ_1, ℓ_2
	$\ell_1 :$	$\mathcal{C}(e_1)$
		branch ℓ_3
	$\ell_2 :$	$\mathcal{C}(e_2)$
		branch ℓ_3
	$\ell_3 :$	

Une (mauvaise) idée

- | Dans le langage source, l'opérateur de flot de contrôle « **if ... then ... else ...** » est très peu **expressif** : il ne permet pas de coder une boucle par exemple.
- | Au contraire, dans le langage de destination, on a **toute liberté** pour coder des itérations :

```
remember 1  
 $\ell$  : remember 1  
add  
branch  $\ell$ 
```

- | Pourquoi ne pas rajouter une construction de branchement arbitraire (**goto**) directement dans le langage source ?

Un langage d'expressions avec **goto**

```
 $e ::= n$   
|  $x$   
|  $e + e$   
|  $e * e$   
| let  $x := e$  in  $e$   
| true | false  
| if  $e$  then  $e$  else  $e$   
|  $x \mathcal{R}^? y$   
|  $\ell : e$   
| goto  $\ell$ 
```

Une sémantique pour le **goto**

- | De façon à pouvoir évaluer l'expression ciblée par un **goto**, une structure associative doit associer une expression à toute étiquette ℓ .
- | On se donne une syntaxe pour représenter cette association :

$$\xi ::= \bullet$$
$$| \quad \xi; \ell \mapsto e$$

- | On suppose qu'une première analyse de l'expression e a conduit à la construction du dictionnaire associant une expression à chaque étiquette.
- | On étend ensuite la syntaxe des jugements d'évaluation :

$$\eta, \xi \vdash e \Downarrow v$$

qui se lit :

« Sous l'environnement η et considérant le dictionnaire ξ , l'évaluation de l'expression e mène à la valeur v . »

Règles de la sémantique à grands pas

$$\frac{, ; \mapsto e \vdash e \Downarrow v}{, \vdash : e \Downarrow v}$$

$$\frac{() = e \quad , \vdash e \Downarrow v}{, \vdash \mathbf{goto} \Downarrow v}$$

Example

```
let  $n := 5$  in  
 $\ell$  : if  $n = 0$  then 1  
    else  $n \times (\text{let } n := n - 1 \text{ in goto } \ell)$ 
```

Discipline de la culture du raisonnement global sur les programmes

- | En présence de **goto**, pour savoir si une expression est bien formée, il ne suffit plus de regarder « au dessus » et de vérifier la bonne liaison des variables.
 - | En effet, si cette expression est étiquetée par ℓ , il faut maintenant vérifier **dans l'ensemble du programme** qu'à toute occurrence de l'expression **goto** ℓ l'environnement d'évaluation définit toutes les variables nécessaires à l'évaluation de la fonction.
- ⇒ C'est une forme de portée dynamique.

Exemple

```
let fact :=  
  let n := 5 in  
    ℓ : if n = 0 then 1  
        else n × (let n := n - 1 in goto ℓ)  
in  
let y := goto ℓ in y
```

- | Cette expression est bloquée car l'évaluation de **goto** ℓ nécessite une variable n qui n'est plus définie.

Di culté du raisonnement global sur les programmes

- | En présence de **goto**, même si une expression est bien formée, il est difficile de s'assurer qu'elle fait bien « ce que l'on veut ».

Exemple

```
let fact5 :=  
  let n := 5 in  
    ℓ : if n = 0 then 1  
        else n × (let n := n - 1 in goto ℓ)  
in  
let y := let n := -5 in goto ℓ in y
```

- | Quand un programme a **plusieurs points d'entrée** , il est difficile de raisonner sur la cohérence des entrées.
 - | Imaginez une expression de plusieurs milliers de lignes de code : pour s'assurer que fact5 se comporte bien, il faut vérifier la totalité du programme.
- ⇒ Ceci est caractéristique d'un logiciel **non modulaire**.

Un autre exemple : un programme BASIC

```
10 INPUT "What is your name: ", U$
20 PRINT "Hello "; U$
30 INPUT "How many stars do you want: ", N
40 S$ = ""
50 FOR I = 1 TO N
60 S$ = S$ + "*"
70 NEXT I
80 PRINT S$
90 INPUT "Do you want more stars? ", A$
100 IF LEN(A$) = 0 THEN GOTO 90
110 A$ = LEFT$(A$, 1)
120 IF A$ = "Y" OR A$ = "y" THEN GOTO 30
130 PRINT "Goodbye "; U$
140 END
```

La crise du logiciel

- | Cette incapacité à raisonner de façon modulaire sur les programmes a provoqué une crise du logiciel dans les années 60.
 - | C'est à cette époque que le coût de la construction du logiciel a dépassé le coût de celle des machines.
 - | Les logiciels étaient alors de faible qualité, inefficaces, incorrects, impossibles à faire évoluer et à corriger, ...
- ⇒ Pour remédier à ces problèmes, a été introduit un ensemble de méthodes et d'outils : **la programmation structurée**.

Les mécanismes de la programmation structurée

Contrôle structuré

- | La programmation structurée interdit une utilisation arbitraire du **goto**.
- | Il a été prouvé que l'on peut se limiter à un ensemble restreint d'opérateur de flot de contrôle :

Théorème (Complétude de la programmation structurée)

Toute fonction calculable peut être réalisée dans un langage de programmation qui combine les sous-programmes de seulement trois façons :

1. *Séquencement : exécuter un sous-programme puis un autre.*
 2. *Sélection : exécuter un sous-programme en fonction d'un booléen.*
 3. *Répétition : exécuter un sous-programme tant qu'une condition est vraie.*
- | Ainsi, on peut interdire le **goto**.

Comment réaliser ces opérateurs de flot de contrôle ?

- I Nous allons voir deux façons de réaliser ces opérateurs de contrôle, l'une basée uniquement sur les **appels de fonctions** (cours d'aujourd'hui) et l'autre sur les opérateurs de flot de contrôle de la **programmation impérative** (un cours prochain).

Les appels de fonctions

Utilité des fonctions

- | Factorisation du code.
- ⇒ À sémantique équivalente, un programme court est **préférable** à un long.
- | Réutilisation du code.
- ⇒ Il vaut mieux raisonner **une fois** de façon générale que d'instancier un raisonnement particulier plusieurs fois.
- | Respecte le principe de **décomposition des problèmes en sous-problèmes**.
- ⇒ C'est l'unique moyen d'aborder un problème **complexe**.
- | Permet de travailler en équipe.
- ⇒ Il suffit de s'entendre sur l'**interface** et la **spécification** des fonctions pour se partager leur développement.
- | Permet d'**abstraire**, c'est-à-dire se doter du **vocabulaire adéquat** pour résoudre le problème. On **cache ainsi les détails d'implémentation non pertinents**, que l'on pourra d'ailleurs **changer indépendamment des utilisations de la fonction** tant que l'on ne change pas la spécification.
- ⇒ Une fois un problème résolu par une fonction, on ne s'intéresse plus à la façon de résoudre ce problème mais à ce qu'apporte sa solution.

Les fonctions de seconde classe

- | Aujourd'hui, nous allons nous intéresser à une implantation particulière des fonctions telle qu'on la trouve dans les **langages de programmation du premier ordre** (C, PASCAL, ...).
- | Dans ces langages du premier ordre (contrairement aux langages fonctionnels ou objets, dits d'ordre supérieur), les fonctions sont des composants logiciels que l'on peut pas manipuler comme des valeurs arbitraires du calcul.
- | En d'autres termes, dans un langage du premier ordre, l'ensemble des fonctions est fixé une fois pour toute par le code source du programme et n'évolue pas au cours de l'évaluation du programme.

Un langage d'expressions et de fonctions

$$\begin{array}{lcl} e & ::= & \dots \\ & | & f(e_1, \dots, e_n) \end{array} \quad (1)$$

$$\textit{declaration} ::= \mathbf{def} \, f(x_1, \dots, x_n) := e \quad (2)$$

$$\textit{programme} ::= \textit{declaration}^+ \mathbf{in} \, e \quad (3)$$

- | (1) représente les appels de fonction.
- | (2) définit une fonction f d'arguments formels x_1, \dots, x_n et de corps e .
- | (3) est un programme composé de déclarations et d'un corps.

Example

```
def fact(n) :=  
  if n = 0 then 1 else n * fact (n - 1)  
in  
  fact (5)
```

Example

```
def even( $n$ ) :=  
  if  $n = 0$  then true else if  $n = 1$  then false else odd ( $n - 1$ )  
def odd( $n$ ) :=  
  if  $n = 0$  then false else if  $n = 1$  then true else even ( $n - 1$ )  
in  
  even (5)
```

Sémantique opérationnelle

- | De nouveau, il suffit d'une première analyse capable d'associer dans un dictionnaire ξ à chaque nom de fonction f , son corps et l'ensemble de ses arguments formels.

Règles de la sémantique opérationnelle

$$\frac{\begin{array}{c} \forall i \in \{1..n\} \quad , \vdash e_i \Downarrow v_i \\ (f) = (x_1, \dots, x_n, e) \quad \bullet; (x_1 \mapsto v_1) \dots; (x_n \mapsto v_n), \vdash e \Downarrow v \end{array}}{\quad , \vdash f(e_1, \dots, e_n) \Downarrow v}$$

Exemple

```
def fact(n) :=  
  if n = 0 then 1 else n * fact (n - 1)  
in  
  fact (3)
```

Exercice

Écrivez l'arbre de dérivation correspondant à l'évaluation de ce programme.

Expressions bloquées

- | Un appel de fonction est bloqué dans deux cas de figure :
 - ▶ le nom de la fonction est indéfini ;
 - ▶ l'arité de la fonction est incorrecte (trop ou trop peu d'arguments effectifs).
- | Par ailleurs, l'évaluation du corps de la fonction appelée peut échouer si les types des arguments effectifs ne sont pas ceux attendus par la fonction.
- | Pour le premier cas, une analyse statique de bonne liaison des noms de fonction dans le dictionnaire suffit.
- | Pour le second cas, et aussi pour s'assurer que les types des arguments effectifs sont corrects, il faut associer à chaque fonction une **signature**.

Signature de fonction

- La syntaxe des signatures de fonction est définie par :

$$\sigma ::= \tau \times \dots \times \tau \rightarrow \tau$$

où τ est soit le type **int**, soit le type **bool**.

- On suppose, pour le moment, qu'il existe une fonction Σ qui associe une signature σ à chaque fonction f
- On peut alors étendre le jugement de typage :

$$\Sigma, \Gamma \vdash e : \tau$$

qui se lit :

« Sous le dictionnaire de signatures Σ et l'environnement de typage Γ , l'expression e a le type τ . »

Règle de typage des appels de fonction

$$\frac{\Sigma(f) = e_1 \times \dots \times e_n \rightarrow \quad \forall i \in \{1..n\} \quad \Sigma, \Gamma \vdash e_i : \tau_i}{\Sigma, \Gamma \vdash f(e_1, \dots, e_n) : \tau}$$

Vérification du dictionnaire de signatures de fonction

- | Il reste maintenant à s'assurer que la signature $\xi(f)$ est effectivement une signature correcte pour f .
- | Pour cela, on définit le jugement :

$$\Sigma \vdash \mathbf{def} \ f(x_1, \dots, x_n) := e : \sigma$$

par l'unique règle :

$$\frac{\Sigma, \bullet; x_1 : \tau_1; \dots; x_n : \tau_n \vdash e : \tau}{\Sigma \vdash \mathbf{def} \ f(x_1, \dots, x_n) := e : \tau_1 \times \dots \times \tau_n \rightarrow \tau}$$

Comment obtenir le dictionnaire des signatures ?

- I Il y a deux procédés pour obtenir le dictionnaire des signatures Σ :
 1. On étend la syntaxe du langage par des **annotations de type**, ainsi :

$$\textit{declaration} ::= \textbf{def } f(x_1 : \tau_1, \dots, x_n : \tau_n) : \tau := e$$

dont on peut extraire facilement Σ .

2. On utilise un **algorithme d'inférence de type** dont nous parlerons dans un prochain cours.

Caractéristique du mécanisme d'appel de fonction

- | Un appel de fonction garantit que les arguments formels de la fonction ne sont pas modifiables de l'extérieur.
- ⇒ C'est cette propriété qui permet de garantir la **modularité**.

Nous regarderons les choses d'un peu plus près . . .

- | La sémantique opérationnelle à grands pas ne met pas en lumière le **mécanisme de va-et-vient du contrôle** entre la fonction appelée et le contexte appelant.
 - | Il faut pourtant le comprendre pour écrire un compilateur.
- ⇒ Ce sera le sujet du prochain cours.

Synthèse

Synthèse

- | Nous avons introduit une analyse statique appelée typage.
- | Tous les programmes bien typés s'évaluent sans bloquer.
- | Nous avons une sémantique du branchement conditionnel et sa compilation.
- | Un autre opérateur de flot de contrôle très riche a été présenté : le **goto**.
- | Cependant, il s'avère **trop** riche.
- | Nous avons abordé le mécanisme d'appel de fonction.

INTRODUCTION À LA COMPILATION

Cours 10 : Expérimentations autour des fonctions

Yann Régis-Gianas
`yrg@pps.jussieu.fr`

PPS - Université Denis Diderot – Paris 7

Compilation des appels de fonction vers une machine virtuelle

Problématique

- | Dans ce cours, nous supposons que les programmes sont bien typés.
- | Un appel $f\ e_1, \dots, e_n$ réalise un **va-et-vient** du flot de contrôle :
 1. L'**appelant** donne le contrôle au corps de la fonction **appelée** f .
 2. Le corps de la fonction f est évalué.
 3. Le contrôle est rendu à l'appelant.
- | Pour compiler ce mécanisme, l'instruction de branchement va être utile.

Rappel du langage de la machine abstraite à pile

programme

ℓ *instruction* $^+$

instruction

remember n

$n \in \mathbb{N}$

add | **mul** | **div** | **sub**

cmple | **cmpge** | **cmpeq**

cmplt | **cmpgt**

getvar i

$i \in \mathbb{N}$

define

undefine

branch ℓ | **branchif** ℓ, ℓ

Rappel du langage source

$$\begin{array}{c} e \\ | \\ \dots \\ f \ e_1, \dots, e_n \end{array} \quad (1)$$

$$\textit{declaration} \quad \mathbf{def} \ f \ x_1, \dots, x_n \quad e \quad (2)$$

$$\textit{programme} \quad \textit{declaration}^+ \ \mathbf{in} \ e \quad (3)$$

Exemple de traduction incorrecte

```
| C def succ x      x  1 in succ succ 40    =  
    remember 40    ; Pousse 40 sur la pile des résultats intermédiaires.  
    define         ; Déplace 40 sur la pile des variables.  
    branch  $\ell_{succ}$  ; Exécute le corps de la fonction succ.  
    define         ; Déplace le résultat sur la pile des variables.  
    branch  $\ell_{succ}$    ; Exécute de nouveau le corps de la fonction succ.  
 $\ell_{succ}$  remember 1  ; Pousse 1 sur la pile.  
    getvar 0        ; Récupère la valeur de x.  
    add             ; Effectue l'addition.
```

Exercice

Cette traduction est, bien sûr, incorrecte. Comment la corriger ?

Exemple de traduction correcte

```
| C def succ x      x  1 in succ succ 40      =  
    remember 40      ; Pousse 40 sur la pile des résultats intermédiaires.  
    define          ; Déplace 40 sur la pile des variables.  
    remember  $\ell_1$     ; Pousse l'étiquette de continuation du calcul.  
    branch  $\ell_{succ}$     ; Exécute le corps de la fonction succ.  
 $\ell_1$  : define          ; Déplace le résultat sur la pile des variables.  
    remember  $\ell_2$     ; Pousse l'étiquette de continuation du calcul.  
    branch  $\ell_{succ}$     ; Exécute de nouveau le corps de la fonction succ.  
 $\ell_2$  : exit           ; Stoppe le calcul.  
 $\ell_{succ}$  : remember 1  ; Pousse 1 sur la pile.  
    getvar 0          ; Récupère la valeur de  $x$ .  
    add               ; Effectue l'addition.  
    undefine          ; L'argument  $x$  est maintenant indéfini.  
    swap              ; Pousse le second élément de  $\zeta_r$  à son sommet.  
    ubranch           ; Saute à l'étiquette au sommet de  $\zeta_r$ .
```

Traduction

```
| C f e1, ..., en  
    C e1  
    define  
    ...  
    C en  
    define  
    remember  $\ell$   
    branch  $\ell_f$   
     $\ell$  ...  
| C def f x1, ..., xn e1 in e2  
    C e2  
    exit  
     $\ell_f$  C e1  
    undefine  
    ... (n fois) ...  
    undefine  
    swap  
    ubbranch
```


Extension du langage de la machine virtuelle

- 4 extensions du langage de la machine abstraite sont nécessaires :
 1. On peut pousser des étiquettes sur la pile (**remember** ℓ).
 2. **ubbranch** est un **branchement à une adresse inconnue** fournie au sommet de ζ_r .
 3. **swap** échange les deux éléments au sommet de ζ_r .
 4. **exit** stoppe la machine.

Deux remarques sur l'utilisation des piles

- | Il y a beaucoup de travail au sommet des deux piles :
 1. Chaque définition de variable provoque un empilement et un dépilement.
 2. Chaque résultat intermédiaire aussi.
- | À chaque empilement, on alloue un petit espace mémoire.
- | À chaque dépilement, on désalloue un petit espace mémoire.

Exercice

Peut-on pré-allouer l'espace de pile nécessaire à l'évaluation d'une expression e ?
(Astuce : observez la fonction de compilation.)

Calcul de l'espace de pile de variables nécessaire

- On définit la fonction $s_v e$ correspondant au nombre maximal de variables nécessaires à l'évaluation de l'expression e (en ne suivant pas les appels de fonctions) :

$s_v x$	0
$s_v n$	0
$s_v e_1 \otimes e_2$	$\max s_v e_1, s_v e_2$
$s_v \text{ let } x = e_1 \text{ in } e_2$	$\max s_v e_1, 1 + s_v e_2$
$s_v \text{ if } e_c \text{ then } e_1 \text{ else } e_2$	$\max s_v e_c, s_v e_1, s_v e_2$
$s_v f e_1, \dots, e_n$	$\max n, \max_{i \in \{1..n\}} s_v e_i$

Calcul de l'espace de pile de résultats nécessaire

- On définit la fonction $s_r\ e$ correspondant au nombre maximal de résultats temporaires qui peuvent apparaître durant l'évaluation de l'expression e (en ne suivant pas les appels de fonctions) :

$s_v\ x$	1
$s_v\ n$	1
$s_v\ e_1 \otimes e_2$	$\max\ s_r\ e_1, 1 + s_r\ e_2$
$s_r\ \text{let } x = e_1\ \text{in } e_2$	$\max\ s_r\ e_1, s_r\ e_2$
$s_r\ \text{if } e_c\ \text{then } e_1\ \text{else } e_2$	$\max\ s_r\ e_c, s_r\ e_1, s_r\ e_2$
$s_r\ f\ e_1, \dots, e_n$	$1 + \max_{i \in \{1..n\}} s_r\ e_i$

Modification de la machine virtuelle

- | On introduit quatre instructions :
 - ▶ **alloc_vstack** N : alloue un bloc de taille N au sommet de la pile ζ_v
 - ▶ **alloc_rstack** N : alloue un bloc de taille N au sommet de la pile ζ_r
 - ▶ **free_vstack** : désalloue le bloc au sommet de la pile ζ_v .
 - ▶ **free_rstack** : désalloue le bloc au sommet de la pile ζ_r .
- | L'implémentation des empilements et des dépilements est simplifiée.
- | Reste une question :

Où placer ces préallocations et ces désallocations ?

Première tentative

- | Une façon sûre de procéder consiste à les rajouter avant chaque empilement et dépilement.
- | On ne gagne alors rien vis-à-vis du mécanisme précédent.

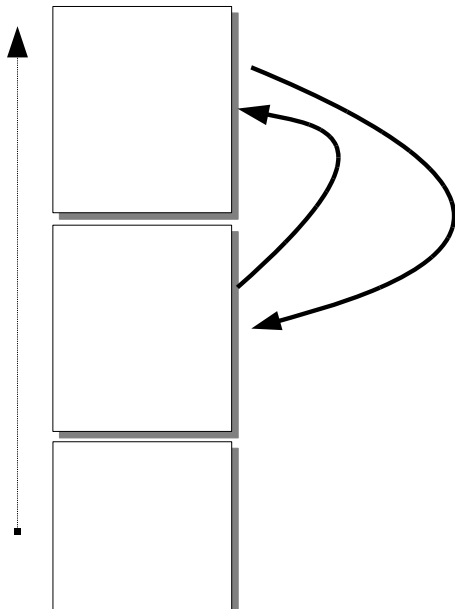
Seconde tentative

- | **Avant un appel de fonction** de la forme $f \ e_1, \dots, e_n$:
 - ▶ On évalue e_1, \dots, e_n , ce qui empile des valeurs v_1, \dots, v_n au sommet de ζ_r .
 - ▶ On pré-alloue un bloc de taille $n + s_v(e)$ (où e est le corps de f) au sommet de ζ_v et un bloc de taille $s_r(e)$ au sommet de ζ_r .
 - ▶ On déplace les valeurs v_1, \dots, v_n du second bloc de ζ_r au premier bloc de ζ_v .
- | **À la sortie de la fonction** :
 - ▶ On déplace la valeur au sommet du premier bloc ζ_r vers le sommet du second bloc de ζ_r .
 - ▶ On désalloue ces deux blocs.
- | On a la garantie que le corps de la fonction ne modifiera que ces deux blocs au cours de son évaluation.

Bloc d'activation

- | En fait, on peut maintenant **factoriser les deux piles** en une seule.
- | Les deux blocs pré-alloués par le mécanisme précédent se fusionnent en un unique bloc appelé **bloc d'activation d'une fonction** qui contient deux sous-blocs : le bloc des variables et le bloc des résultats temporaires.
- | **Ces deux sous-blocs ont des positions connues** dans le bloc d'activation.
- | Les instructions travaillent désormais dans le sous-bloc qui les concernent.
- | On peut en profiter pour allouer un emplacement dans le bloc d'activation pour y stocker l'adresse de continuation du calcul.

Bloc d'activation : vue globale



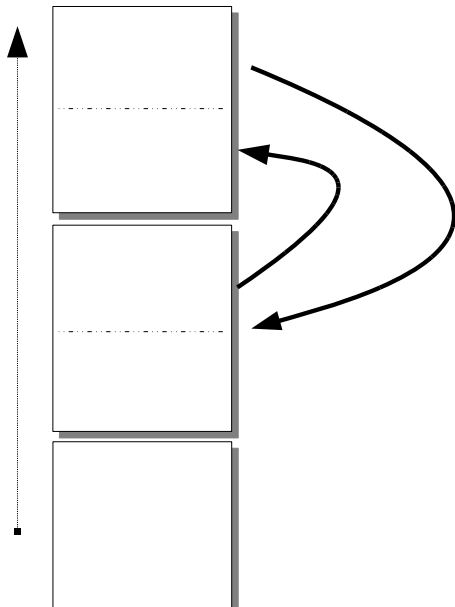
Comment agencer les deux sous-blocs ?

- I Il y a une certaine liberté quant à l'agencement des deux sous-blocs à l'intérieur du bloc d'activation :
 - ▶ Le bloc de variable au-dessus et le bloc de temporaire en dessous.
 - ▶ Le bloc de variable au-dessous et le bloc de temporaire en dessous.

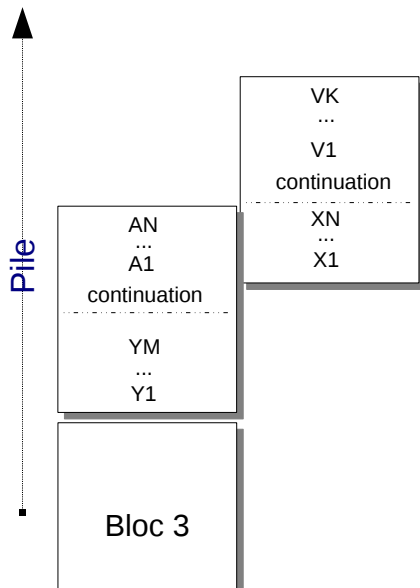
Exercice

En fait, il y a une solution plus astucieuse que l'autre. Laquelle ?

Bloc d'activation : vue interne



Bloc d'activation : l'astuce



Les avantages de cet agencement

- | Quand on place le bloc des variables au dessous du bloc des résultats :
 - ▶ on peut implémenter le passage des arguments effectifs par une simple incrémentation de l'entier représentant le sommet de la pile ;
 - ▶ ce n'est plus la peine de borner l'espace utilisable pour les résultats intermédiaires.
- | Il est encore nécessaire de recopier la valeur de retour de la fonction.

Langage de la machine virtuelle

<i>programme</i>	ℓ	<i>instruction</i> ⁺
<i>instruction</i>		remember v
		add mul div sub
		cmple cmpge cmpeq
		cmplt cmpgt
		getvar i $i \in \mathbb{N}$
		define
		undefine
		branch ℓ branchif ℓ, ℓ
		ubbranch
		alloc_stack N, M $N, M \in \mathbb{N}$
		shift N $N \in \mathbb{N}$
		unshift N $N \in \mathbb{N}$
		return
v		n $n \in \mathbb{N}$
		ℓ

Sémantique des nouvelles instructions

- | **alloc_stack** N, M :

Alloue un bloc d'activation de taille M débutant par un espace réservé pour N variables locales.

- | **shift** K :

Déplace la base du bloc d'activation courant de K emplacements vers le bas de façon à capturer le sommet du bloc d'activation précédent, où se trouvent la valeur des arguments effectifs.

- | **unshift** K :

Transfère K valeurs au sommet du bloc d'activation courant à la base de ce même bloc puis déplace la base du bloc d'activation courant de K emplacements vers le haut de façon à transférer le retour de la fonction appelée vers son appelant.

- | **return** :

Récupère l'adresse de retour ℓ dans le bloc d'activation courant, désalloue ce bloc et saute à l'adresse ℓ .

Traduction

| Soit $M \equiv s_v \ e_1 \quad K \quad s_r \ e_1, N \equiv s_v \ e_1$

| $C \ f \ a_1, \dots, a_K$

$C \ a_1$

\dots

$C \ a_K$

alloc_stack M, N

remember ℓ

shift K

branch ℓ_f

$\ell \quad \dots$

| $C \ \mathbf{def} \ f \ x_1, \dots, x_n \quad e_1 \ \mathbf{in} \ e_2$

$C \ e_2$

exit

$\ell_f \quad C \ e_1$

unshift 1

return

Calcul statique des indices

- | Les instructions qui empilent et dépilent des variables doivent encore incrémenter ou décrémenter l'indice du sommet de la pile des variables dans le sous-bloc.

Exercice

Peut-on pré-calculer ces indices ?

Cas des appels récursifs

```
def fact  $n$  int  
    if  $n = 0$  then 1 else  $n \times \text{fact } n - 1$   
in fact 3
```

Exercice

Compiler ce programme et exécuter le code compilé obtenu.

Un compilateur optimisant

- | On peut modifier la fonction de compilation pour qu'elle produise un programme qui **réutilise** le bloc d'activation de la fonction appelante pour évaluer le corps de la fonction appelée.
- | Il faut s'assurer :
 - ▶ que la continuation de ce bloc reste celle de l'appelant ;
 - ▶ que le bloc d'activation ait d'une taille suffisante. . .

Exercice (un peu difficile)

Définissez cette fonction de compilation.

Peut-on faire mieux ?

- | Nous avons choisi de pré-allouer les blocs d'activation au niveau des points d'entrée et de sortie des fonctions.
 - | Ne peut-on pas pré-allouer les blocs d'une façon plus globale ?
- ⇒ Ce n'est pas toujours possible !
- | Exemple : la fonction factorielle.

Exercice

Imaginez des situations où cette optimisation **interprocédurale** est applicable.

Les fonctions comme valeur de première classe

Une motivation : des structures de contrôle génériques

```
def repeat  $f$  int  $\times$  int  $\rightarrow$  int,  $accu$  int,  $n$  int  
  if  $n = 0$  then  $accu$  else repeat  $f$   $n$ ,  $accu$ ,  $f$ ,  $n - 1$   
in repeat 1, fun  $x$  int,  $y$  int  $\Rightarrow x \times y$ , 3
```

Une motivation : des structures de contrôle génériques

```
def repeat  $f$  int  $\times$  int  $\rightarrow$  int,  $accu$  int,  $n$  int  
  if  $n = 0$  then  $accu$  else repeat  $f$   $n$ ,  $accu$ ,  $f$ ,  $n - 1$   
in repeat 1, fun  $x$  int,  $y$  int  $\Rightarrow x \times y$ , 3
```

- | Grâce à cette **fonction d'ordre supérieur**, on peut coder une boucle **for** !

Changement de syntaxe

e	n
	x
	$e \otimes e$
	let $x \ \tau \quad e$ in e
	true false
	if e then e else e
	$x \ \mathcal{R}^? y$
	fun $x_1 \ \tau_1, \dots, x_n \ \tau_n \Rightarrow e \quad (1)$

- | (1) est une **fonction anonyme** dont les arguments formels sont x_1, \dots, x_n et dont le corps est l'expression e .

Nouvelles règles (incorrectes) d'évaluation

$$\underline{\xi_0(f) = (x_1, \dots, x_n, e) \quad ; \quad (x_1 \quad \eta, \xi_{i-1} \quad e_i \quad v_i, \xi_i \quad v_1) \dots ; (x_n \quad v_n), \xi_n \quad e \quad v, \xi'}$$

Boom !

```
let add1
  let x = 1 in
    fun y : int => x + y
in
  add1 1
```

- | L'évaluation de la fonction « **fun** *y* : **int** \Rightarrow *x* + *y* » échoue car l'environnement n'a pas de valeur associée à *x* !

Synthèse

Synthèse

- | Nous avons compilé efficacement les appels de fonction d'un langage du premier ordre vers une machine abstraite à pile.
 - | Le passage à l'ordre supérieur semble délicat. . .
- ⇒ Nous l'étudierons lors du prochain cours.

INTRODUCTION À LA COMPILATION

Cours 11 : Les fonctions de première classe

Yann Régis-Gianas
`yrg@pps.univ-paris-diderot.fr`

PPS - Université Denis Diderot – Paris 7

Enquête

Boom !

```
let add1 :=  
  let x := 1 in  
    fun (y : int) ⇒ x + y  
in  
  add1 (1)
```

Exercice

Comment évaluer ce programme ?

La réponse « symbolique »

- | On détermine la valeur associée à `add1` par substitution du `let` interne. :

```
let add1 :=  
  fun (y : int) ⇒ 1 + y  
in  
  add1 (1)
```

- | On substitue cette valeur à l'unique occurrence de `add1` :

```
(fun (y : int) ⇒ 1 + y) (1)
```

- | L'évaluation de l'appel de fonction anonyme consiste à substituer ses arguments effectifs aux arguments formels :

```
1 + 1
```

- | On obtient finalement la valeur « 2 » par évaluation de « + ».

Problème d'un interprète de la sémantique à petits pas

- | Cependant, l'évaluation à petits pas ne fournit pas un interprète efficace : on réécrit le programme sans arrêt et on doit parcourir ce dernier pour appliquer les substitutions.
 - | Les interprètes efficaces des précédents cours produisaient des valeurs par une inspection linéaire de l'arbre de syntaxe.
- ⇒ Comment représenter du code (exécutable) sous la forme de valeurs ?

Deux approches

1. Une unification des valeurs et des programmes : les S-expressions de LISP.
2. Des valeurs pour représenter du code clos : les fermetures.

LISP

Les S-expressions

- | La syntaxe des expressions de LISP :

$$\begin{array}{lcl} Sexp & ::= & atom \\ & | & (Sexp . Sexp) \end{array}$$

- | Cette syntaxe est aussi celle des valeurs !

Type d'expression

| On classifie les expressions :

- ▶ `cons` : une paire formée de deux expressions.
 - ▶ On appelle `car` la première composante ;
 - ▶ et `cdr` la seconde.¹
- ▶ `int` : un entier.
- ▶ `sym` : un symbole.
- ▶ `prim` : une fonction primitive.
- ▶ `comp` : une fonction anonyme.
- ▶ `spec` : une forme spéciale (une fonction qui n'évalue pas tout ses arguments).

1. `car` pour *Contents of Address Register* et `cdr` pour *Contents of Decrement Register* ◀ ≡ ▶ ≡ ↺ 🔍 ↻

Boucle d'interaction de LISP

read-eval-print

- | La boucle d'interaction de LISP se décompose en trois fonctions :
 - ▶ `read` : transforme une chaîne de caractères en S-expression ;
 - ▶ `eval` : transforme une S-expression en sa forme évaluée, une S-expression ;
 - ▶ `print` : affiche une S-expression sous la forme d'une chaîne de caractères.

⇒ "`read`" et "`print`" sont particulièrement simples à réaliser, reste "`eval`".

Exemples

;; Cette expression s'évalue en 3. "+" est une primitive.

```
(+ 1 2)
```

;; Cette expression définit un symbole x valant 1.

```
(define x 1)
```

;; Une fonction anonyme à deux arguments.

```
(lambda (x y) (+ x y))
```

;; La fonction identité appliquée à la liste vide.

```
((lambda (x) x) ())
```

;; "if" est une forme spéciale : seule une branche est évaluée.

```
(if (= x 2) (1 + 2) (3 + 4))
```

;; "quote" permet de retarder l'évaluation d'une expression

```
((lambda (x) (car (cdr x))) ('(1 2 3 4)))
```


Environnements d'interprétation des S-expressions

- | LISP utilise deux environnements :
 - ▶ un environnement lexical Γ qui associe des S-expressions à des symboles ;
 - ▶ un environnement global Ξ qui enregistre les définitions de fonctions.

Interprétation des S-expressions

- 1 L'évaluation est définie par cas sur le type de S-expressions à évaluer :

$$\begin{aligned}\forall e \in \{\text{int}, \text{spec}, \text{comp}, \text{prim}\} \quad & \text{eval}(e) = e \\ \forall e \in \{\text{sym}\} \quad & \text{eval}(e) = \Gamma(e) \\ \forall e \in \{\text{cons}\} \quad & \text{eval}(e) = \text{apply}(\text{eval}(\text{car}(e)), \text{cdr}(e))\end{aligned}$$

où `apply` est aussi définie par cas sur le type de son premier argument :

$$\begin{aligned}\forall f \in \{\text{comp}, \text{prim}\} \quad & \text{apply}(f, (e_1 \dots e_n)) = \text{eval}(\text{body}(f)) \\ & \text{dans } \Gamma' \equiv ((x_1 \text{ eval}(e_1) \dots (x_n \text{ eval}(e_n)) \Gamma) \\ & \text{où } \text{args}(f) = x_1 \dots x_n \\ \forall f \in \{\text{sym}\} \quad & \text{apply}(f, (e_1 \dots e_n)) = \text{apply}(\Xi(f), (e_1 \dots e_n)) \\ \forall f \in \{\text{spec}\} \quad & \text{apply}(f, (e_1 \dots e_n)) = \text{eval}(\text{body}(f)) \\ & \text{dans } \Gamma' \equiv ((x_1 e_1) \dots (x_n e_n) \Gamma) \\ & \text{où } \text{args}(f) = x_1 \dots x_n\end{aligned}$$

Interprète LISP en O'Caml

Exercice du cahier de vacances.

Écrivez un évaluateur de LISP en O'Caml.

Interprète LISP en LISP (de Paul Graham)

```
(defun null. (x)  
  (eq x '()))
```

```
(defun and. (x y)  
  (cond (x (cond (y 't) ('t '())))  
        ('t '()))))
```

```
(defun not. (x)  
  (cond (x '())  
        ('t 't)))
```

```
(defun append. (x y)  
  (cond ((null. x) y)  
        ('t (cons (car x) (append. (cdr x) y)))))
```

```
(defun list. (x y)  
  (cons x (cons y '())))
```

Interprète LISP en LISP (par Paul Graham)

```
(defun pair. (x y)
  (cond ((and. (null. x) (null. y)) '())
        ((and. (not. (atom x)) (not. (atom y)))
         (cons (list. (car x) (car y))
                 (pair. (cdr x) (cdr y))))))

(defun assoc. (x y)
  (cond ((eq (caar y) x) (cadar y))
        ('t (assoc. x (cdr y)))))
```

Interprète LISP en LISP (par Paul Graham)

```
(defun eval. (e a)
  (cond
    ((atom e) (assoc. e a))
    ((atom (car e))
     (cond
       ;; Special forms
       ((eq (car e) 'quote) (cadr e))
       ((eq (car e) 'atom) (atom (eval. (cadr e) a)))
       ((eq (car e) 'eq) (eq (eval. (cadr e) a) (eval. (caddr e) a)))
       ((eq (car e) 'car) (car (eval. (cadr e) a)))
       ((eq (car e) 'cdr) (cdr (eval. (cadr e) a)))
       ((eq (car e) 'cons) (cons (eval. (cadr e) a) (eval. (caddr e) a)))
       ((eq (car e) 'cond) (evcon. (cdr e) a))
       ('t (eval. (cons (assoc. (car e) a) (cdr e)) a))))
    ;; Anonymous function
    ((eq (caar e) 'lambda)
     (eval. (caddar e) (append. (pair. (cadar e) (evlis. (cdr e) a)) a)))))
```

Interprète LISP en LISP (par Paul Graham)

```
(defun evcon. (c a)
  (cond ((eval. (caar c) a)
        (eval. (cadar c) a))
        ('t (evcon. (cdr c) a))))
```

```
(defun evlis. (m a)
  (cond ((null. m) '())
        ('t (cons (eval. (car m) a) (evlis. (cdr m) a)))))
```

Portée dynamique

```
> (define foo 1)
foo
> (define add-foo (lambda (x) (+ x foo)))
add-foo
> ((lambda (foo) (add-foo 3)) 5)
4
```

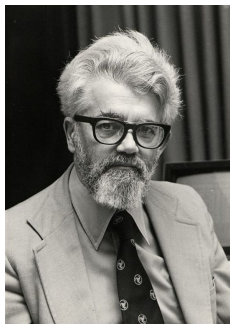
```
> (define foo 1)
foo
> (define add-foo (lambda (x) (+ x foo)))
add-foo
> ((lambda (foo) (add-foo 3)) 5)
8
```

Exercice

Parmi ces deux comportements, lequel correspond à notre interprète ?

Un peu d'histoire – retour en 1960

*"Recursive Functions of Symbolic Expressions
and
Their Computation by Machine, Part I"*



J. McCarthy

- | LISP est le langage de haut niveau le plus ancien après FORTRAN.
- | Il a été un véritable laboratoire pour les premières expérimentations des traitements « symboliques » de l'information (gestion de connaissance, logique, transformation et analyse de programmes ...).

Les LISPes modernes

- | LISP vit toujours !
- | Ses descendants sont SCHEME, COMMON-LISP, E-LISP, CLOJURE...
- | Les idées directrices de ces langages sont :
 - ▶ La **dynamicité** :
Le programmeur peut toujours évaluer n'importe quelle S-expression.
Aucun système de type ne restreint l'expressivité du langage.²
 - ▶ La **reflexivité** :
Les programmes sont des S-expressions comme les autres.
On peut certes les évaluer mais aussi les observer, transformer, calculer...
 - ▶ La **généralité** :
Comme tous les langages fonctionnels, on cherche à écrire des fonctions les plus générales possibles pour augmenter la réutilisabilité.
- | LISP a aussi évolué :
 - ▶ On utilise maintenant par défaut la **portée lexicale statique**.
 - ▶ Les implémentations modernes de LISP fournissent un **compilateur**.

2. Cependant, des langages comme SCHEME proposent de plus en plus d'analyse statique pour aider le programmeur à ne pas faire d'erreurs stupides...

Du λ -calcul aux fermetures

Introduction

- | Problème dans la théorie des ensembles (de l'époque) :

$$y = \{x \mid x \notin x\}$$

B. Russell



A. Church

- | Autre théorie des ensembles centrée sur les fonctions :

Le λ -calcul

Syntaxe du λ -calcul

$e ::=$		Expression
	x, f, \dots	<i>Variable</i>
	\mathbf{c}	<i>Constante $\mathbf{c} \in \mathbb{C}$</i>
	$e\ e$	<i>Application</i>
	$\lambda x. e$	<i>Fonction</i>
$a, v ::=$		Valeurs
	x, f, \dots	<i>Variable</i>
	\mathbf{c}	<i>Constante $\mathbf{c} \in \mathbb{C}$</i>
	$\lambda x. e$	<i>Fonction</i>

- Le λ (le mot-clé fun) joue un rôle de **lieur** (comme le mot-clé let).

Exercice

Définissez l'ensemble des variables libres d'une expression e .

Syntaxe du λ -calcul

$e ::=$		Expression
	x, f, \dots	Variable
	\mathbf{c}	Constante $\mathbf{c} \in \mathbb{C}$
	$e\ e$	Application
	$\text{fun } x \Rightarrow e$	Fonction
$a, v ::=$		Valeurs
	x, f, \dots	Variable
	\mathbf{c}	Constante $\mathbf{c} \in \mathbb{C}$
	$\text{fun } x \Rightarrow e$	Fonction

- Le λ (le mot-clé fun) joue un rôle de **lieur** (comme le mot-clé let).

Exercice

Définissez l'ensemble des variables libres d'une expression e .

Associativité et priorité

- | « $e_1 \ e_2 \ e_3$ » se lit « $(e_1 \ e_2) \ e_3$ »
- | « $\text{fun } x \Rightarrow e_1 \ e_2$ » se lit « $\text{fun } x \Rightarrow (e_1 \ e_2)$ ».

Définition des variables libres

$$\begin{array}{lll} FV(x) & = & \{x\} \\ FV(n) & = & \emptyset \\ FV(\text{fun } x \Rightarrow e) & = & FV(e) \setminus \{x\} \\ FV(e_1 \ e_2) & = & FV(e_1) \cup FV(e_2) \end{array} \quad n \in \mathbb{N}$$

Exercice

Comment définir le mécanisme de substitution sur ce langage ?

Substitution (fausse)

$$\begin{array}{lll} x\{y \mapsto e\} & = & x \qquad \text{si } x \neq y \\ x\{y \mapsto e\} & = & e \qquad \text{si } x = y \\ n\{y \mapsto e\} & = & n \qquad n \in \mathbb{N} \\ (\text{fun } x \Rightarrow e')\{y \mapsto e\} & = & \text{fun } x \Rightarrow (e'\{y \mapsto e\}) \\ (e_1 \ e_2)\{y \mapsto e\} & = & e_1\{y \mapsto e\} \cup e_2\{y \mapsto e\} \end{array}$$

Exercice

Par analogie avec la substitution déjà définie pour les expressions avec `let`s, pourquoi cette définition est-elle incorrecte ?

Premier exemple

$$(\text{fun } x \Rightarrow x)\{x \mapsto 1\} = \text{fun } x \Rightarrow (x\{x \mapsto 1\})$$

Premier exemple

$$(\text{fun } x \Rightarrow x)\{x \mapsto 1\} = \text{fun } x \Rightarrow (x\{x \mapsto 1\})$$

- | À gauche : la fonction identité.
- | À droite : la fonction constante égale à 1.

Second example

$$(\text{fun } y \Rightarrow x + y)\{x \mapsto y\} = \text{fun } y \Rightarrow ((x + y)\{x \mapsto y\})$$

Second example

$$(\text{fun } y \Rightarrow x + y)\{x \mapsto y\} = \text{fun } y \Rightarrow ((x + y)\{x \mapsto y\})$$

- | À gauche : la fonction qui ajoute x à son argument.
- | À droite : la fonction qui calcule le double de son argument.

Substitution sans capture

$$\begin{array}{llll} x\{y \mapsto e\} & = & x & \text{si } x \neq y \\ x\{y \mapsto e\} & = & e & \text{si } x = y \\ n\{y \mapsto e\} & = & n & n \in \mathbb{N} \\ (\text{fun } x \Rightarrow e')\{y \mapsto e\} & = & \text{fun } x \Rightarrow (e'\{y \mapsto e\}) & \text{si } x \neq y \text{ et } x \notin FV(e) \\ (e_1 \ e_2)\{y \mapsto e\} & = & e_1\{y \mapsto e\} \cup e_2\{y \mapsto e\} & \end{array}$$

Sémantique à petits pas

$$(\text{-RÉDUCTION}) \frac{}{(\text{fun } x \Rightarrow e_1) e_2 \rightarrow e_1 \{x \mapsto e_2\}}$$

$$(\text{-RÉDUCTION}) \frac{}{c \ v_1 \dots v_n \rightarrow \delta_c(v_1, \dots, v_n)}$$

$$\frac{e_1 \rightarrow e'_1}{e_1 \ e_2 \rightarrow e'_1 \ e_2}$$

$$\frac{e_2 \rightarrow e'_2}{e_1 \ e_2 \rightarrow e_1 \ e'_2}$$

$$\frac{e \rightarrow e'}{\text{fun } x \Rightarrow e \rightarrow \text{fun } x \Rightarrow e'}$$

Exercice

Quelles sont les règles de calculs ? Les règles de passage au contexte ?

Codage du let

- On peut maintenant définir la construction « let ... in ... » comme un sucre syntaxique, c'est-à-dire une construction extérieur au langage noyau.

$$\text{let } x = e_1 \text{ in } e_2 \equiv (\text{fun } x \Rightarrow e_2) e_1$$

Quelques constantes utiles

$0, 1, \dots$	
$+$	avec $\delta_+(n_1, n_2) = n_1 +_{\mathbb{N}} n_2, \dots$
$\text{true}, \text{false}$	
if	avec $\delta_{\text{if}}(\text{true}, v_1, v_2) = v_1$ et $\delta_{\text{if}}(\text{false}, v_1, v_2) = v_2$
fix	avec $\delta_{\text{fix}}(v) = v (\text{fix } v)$

- En fait, ces constantes sont **définissables** en λ -calcul.

(Cours de sémantique de M1)

La nécessité d'une stratégie d'évaluation

- | Le calcul défini par la sémantique à petits pas précédente est très général.
- ⇒ Ses propriétés seront l'objet du cours de sémantique de M1.
- | Cependant, les langages de programmation fonctionnelle utilisent des règles de calcul moins générales, dirigées par **stratégie d'évaluation** dans le but d'améliorer son efficacité (en pratique) et de simplifier la compilation.
- | Par exemple, il est interdit d'évaluer sous le corps des fonctions en dehors des appels de fonctions. De cette façon, on peut compiler une fois le corps de chaque fonction.
- | De plus, une stratégie d'évaluation fournit un procédé **déterministe** pour décider de façon univoque la façon de poursuivre l'évaluation d'un terme.

Quelques stratégies importantes

- | Réduire le redex le plus externe.
- ⇒ Stratégie LMOM (*LeftMost OuterMost*) ou RMOM (*RightMost OuterMost*)
- | Contraindre le terme de droite d'une β -réduction à être une valeur, ou non.
- ⇒ Stratégie d'appel par valeur ou par nom.

Exercice

Comment spécifier ces stratégies ?

Contexte d'évaluation

- | Il est pratique de spécifier les contextes d'évaluation **syntactiquement**.
- | On définit des **contextes d'évaluation** à l'aide de grammaires.
- | On s'assure que, pour tout contexte, il existe un unique sous-terme « **trou** ».
- | Ce trou sert à spécifier où la réduction doit avoir lieu.
- | On **décompose** toute expression e sous la forme d'un contexte d'évaluation C et d'un sous-terme réductible e' prenant la place du trou, ce que l'on note « $C [e']$ ».

Sémantique à petits pas avec contexte d'évaluation

$(\text{fun } x \Rightarrow e) e'$	\rightarrow	$e\{x \mapsto e'\}$	$(\beta\text{-reduction})$
$C[e]$	\rightarrow	$C[e']$	$(\text{Réduction sous-contexte})$
			$\text{si } e \rightarrow e'$

En appel par valeur, arguments évalués de droite à gauche

- On remplace la β -réduction arbitraire en :

$$(\text{fun } x \Rightarrow e) v \rightarrow e\{x \mapsto v\} \quad (\beta_v\text{-reduction})$$

- On définit les contextes d'évaluation :

\mathcal{C}	$::=$	Contexte d'évaluation
	\square	<i>Trou</i>
	$\mathcal{C} e$	<i>Réduction à gauche</i>
	$v \mathcal{C}$	<i>Réduction à droite</i>

- Exemples de décomposition :

- ▶ « $(\text{fun } x \Rightarrow (\text{fun } y \Rightarrow y) 1) ((\text{fun } z \Rightarrow z) 1)$ »
- ▶ « $((\text{fun } x \Rightarrow \text{fun } y \Rightarrow y) 1) 42$ »

En appel par valeur, arguments évalués de droite à gauche

- On remplace la β -réduction arbitraire en :

$$(\text{fun } x \Rightarrow e) v \rightarrow e\{x \mapsto v\} \quad (\beta_v\text{-reduction})$$

- On définit les contextes d'évaluation :

C	$::=$	Contexte d'évaluation
	\square	<i>Trou</i>
	$C e$	<i>Réduction à gauche</i>
	$v C$	<i>Réduction à droite</i>

- Exemples de décomposition :

- ▶ « $(\text{fun } x \Rightarrow (\text{fun } y \Rightarrow y) 1) ((\text{fun } z \Rightarrow z) 1)$ »

$$\begin{cases} C \equiv (\text{fun } x \Rightarrow (\text{fun } y \Rightarrow y) 1) \square \\ e \equiv (\text{fun } z \Rightarrow z) 1 \end{cases}$$

- ▶ « $((\text{fun } x \Rightarrow \text{fun } y \Rightarrow y) 1) 42$ »

En appel par valeur, arguments évalués de droite à gauche

- On remplace la β -réduction arbitraire en :

$$(\text{fun } x \Rightarrow e) v \rightarrow e\{x \mapsto v\} \quad (\beta_v\text{-reduction})$$

- On définit les contextes d'évaluation :

C	$::=$	Contexte d'évaluation
	\square	<i>Trou</i>
	$C e$	<i>Réduction à gauche</i>
	$v C$	<i>Réduction à droite</i>

- Exemples de décomposition :

- ▶ « $(\text{fun } x \Rightarrow (\text{fun } y \Rightarrow y) 1) ((\text{fun } z \Rightarrow z) 1)$ »

$$\begin{cases} C \equiv (\text{fun } x \Rightarrow (\text{fun } y \Rightarrow y) 1) \square \\ e \equiv (\text{fun } z \Rightarrow z) 1 \end{cases}$$

- ▶ « $((\text{fun } x \Rightarrow \text{fun } y \Rightarrow y) 1) 42$ »

$$\begin{cases} C \equiv \square 42 \\ e \equiv (\text{fun } x \Rightarrow \text{fun } y \Rightarrow y) 1 \end{cases}$$

En appel par nom

- Avec la β -réduction classique :

\mathcal{C}	$::=$	Contexte d'évaluation
$\quad $	\square	<i>Trou</i>
$\quad $	$\mathcal{C} e$	<i>Réduction à gauche</i>

- Exemples de décomposition :

- « $(\text{fun } x \Rightarrow (\text{fun } y \Rightarrow y) 1) ((\text{fun } z \Rightarrow z) 1)$ »

$$\begin{cases} \mathcal{C} \equiv \square \\ e \equiv (\text{fun } x \Rightarrow (\text{fun } y \Rightarrow y) 1) ((\text{fun } z \Rightarrow z) 1) \end{cases}$$

- « $((\text{fun } x \Rightarrow \text{fun } y \Rightarrow y) 1) 42$ »

En appel par nom

- Avec la β -réduction classique :

C	$::=$	Contexte d'évaluation
\square		<i>Trou</i>
$C e$		<i>Réduction à gauche</i>

- Exemples de décomposition :

- « $(\text{fun } x \Rightarrow (\text{fun } y \Rightarrow y) 1) ((\text{fun } z \Rightarrow z) 1)$ »

$$\begin{cases} C \equiv \square \\ e \equiv (\text{fun } x \Rightarrow (\text{fun } y \Rightarrow y) 1) ((\text{fun } z \Rightarrow z) 1) \end{cases}$$

- « $((\text{fun } x \Rightarrow \text{fun } y \Rightarrow y) 1) 42$ »

En appel par nom

- Avec la β -réduction classique :

$C ::=$	Contexte d'évaluation
$\quad \quad \square$	<i>Trou</i>
$\quad \quad C e$	<i>Réduction à gauche</i>

- Exemples de décomposition :

- ▶ « $(\text{fun } x \Rightarrow (\text{fun } y \Rightarrow y) 1) ((\text{fun } z \Rightarrow z) 1)$ »

$$\begin{cases} C \equiv \square \\ e \equiv (\text{fun } x \Rightarrow (\text{fun } y \Rightarrow y) 1) ((\text{fun } z \Rightarrow z) 1) \end{cases}$$

- ▶ « $((\text{fun } x \Rightarrow \text{fun } y \Rightarrow y) 1) 42$ »

$$\begin{cases} C \equiv \square 42 \\ e \equiv (\text{fun } x \Rightarrow \text{fun } y \Rightarrow y) 1 \end{cases}$$

Pour la suite...

- | Dans ce cours, nous nous focaliserons sur l'appel par valeur, plus courant.
 - | Vous étudierez l'appel par nom en :
 - ▶ Programmation fonctionnelle avancée (M1).
 - ▶ Programmation comparée (M1).
 - ▶ Sémantique (M1).
- ⇒ Un raffinement, l'évaluation paresseuse, est à la base du langage `HASKELL`.

Sémantique à grands pas pour les valeurs closes

- On définit la relation « $e \Downarrow v$ » qui signifie :
« L'expression **close** e s'évalue en la valeur v . »

$$\frac{}{\text{fun } x \Rightarrow e \Downarrow \text{fun } x \Rightarrow e} \qquad \frac{}{n \Downarrow n}$$
$$\frac{e_1 \Downarrow \text{fun } x \Rightarrow e \quad e_2 \Downarrow a \quad e\{x \mapsto a\} \Downarrow v}{e_1 \ e_2 \Downarrow v}$$

Capture de l'environnement

$$\frac{}{\rho \vdash \text{fun } x \Rightarrow e \Downarrow (\text{fun } x \Rightarrow e)[\rho]}$$

- On étend la syntaxe des valeurs.

$a, v ::=$	Valeurs
$ \quad x, f, \dots$	<i>Variable</i>
$ \quad \mathbf{c}$	<i>Constante $\mathbf{c} \in \mathbb{C}$</i>
$ \quad (\text{fun } x \Rightarrow e)[\rho]$	<i>Fermeture</i>

Sémantique à grand pas : règle de l'application

$$\frac{\rho \vdash e_2 \Downarrow a \quad \rho \vdash e_1 \Downarrow (\text{fun } x \Rightarrow e)[\rho'] \quad \rho' + x \mapsto a \vdash e \Downarrow v}{\rho \vdash e_1 e_2 \Downarrow v}$$

- Quand on veut évaluer le corps d'une fonction, on **rétablit** son environnement d'évaluation.

Expressions bloquées

Exercice

Énumérez quelques expressions qui ne sont pas des valeurs mais qui ne sont pas réductibles.

Système de type simple

- On étend la syntaxe des types du langage des expressions arithmétiques :

τ	$::=$	int
		bool
		$\tau \rightarrow \tau$

Système de type simple pour le λ -calcul

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{}{\Gamma \vdash i : \text{int}} \quad i \in \mathbb{N}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 \ e_2 : \tau_2}$$

$$\frac{\Gamma; (x : \tau_1) \vdash e : \tau_2}{\Gamma \vdash \text{fun } x \Rightarrow e : \tau_1 \rightarrow \tau_2}$$

Un algorithme d'inférence (partielle) de type

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x \Uparrow \tau}$$

$$\frac{}{\Gamma \vdash i \Uparrow \text{int}} \quad i \in \mathbb{N}$$

$$\frac{\Gamma \vdash e_1 \Uparrow \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 \Downarrow \tau_1}{\Gamma \vdash e_1 e_2 \Uparrow \tau_2}$$

$$\frac{\Gamma; (x : \tau_1) \vdash e \Downarrow \tau_2}{\Gamma \vdash \text{fun } x \Rightarrow e \Downarrow \tau_1 \rightarrow \tau_2}$$

$$\frac{\Gamma \vdash e \Uparrow \tau}{\Gamma \vdash e \Downarrow \tau}$$

$$\frac{\Gamma \vdash e \Downarrow \tau}{\Gamma \vdash (e : \tau) \Uparrow \tau}$$

Un système trop restrictif ?

- | Voici un programme :

```
let id = fun x ⇒ x in  
  if id true then id 42 else 21
```

- | Ce programme n'est pas bloqué.
 - | Ce programme est mal typé pour de mauvaises raisons.
 - | En effet, quel type donner à `id` ?
- ⇒ Successivement, le type `bool → bool` et le type `int → int`.

Le polymorphisme à la ML étend
les types des valeurs nommées par des `lets`
à des familles de types, appelées **schémas de type**.

- ⇒ Cours de compilation de M1.

Synthèse

Synthèse

- | Aujourd'hui : la représentation du code comme une valeur de première classe.
- | La prochaine fois :
 - ▶ Comment compiler le —calcul vers une machine virtuelle ?
 - ▶ Comment compiler le —calcul vers un langage du premier ordre ?
 - ▶ Les fermetures pour les langages objets.

INTRODUCTION À LA COMPILATION

Cours 12 : Compilation des fonctions de première classe

Yann Régis-Gianas
yrg@pps.univ-paris-diderot.fr

PPS - Université Denis Diderot – Paris 7

Une machine abstraite pour les langages fonctionnels

Retour à une machine à deux piles

- | Deux mécanismes sont essentiels pour traiter les fermetures :
 - ▶ pour les construire : il faut **capturer** (un fragment de) l'environnement lexical ;
 - ▶ pour les appliquer : il faut **restaurer** cet environnement lexical.
 - | L'évolution de la pile des résultats intermédiaires est maintenant **décorrélée** de celle de l'environnement lexical.
- ⇒ On s'intéresse donc de nouveau une machine à **deux** piles ζ_v et ζ_r .

Instructions liées aux fermetures

- | **close :**

Dépile de ζ_v une pile de variables et un pointeur de code et empile une fermeture qui représente une paire de ces deux composantes.

- | **open :**

Dépile une fermeture composée d'un pointeur de code pc et d'une pile ζ'_v , pousse la pile ζ'_v et pc au sommet de ζ_r .

Compilation du λ -calcul vers la machine virtuelle

| $\mathcal{C}(\Gamma, x) = \text{getvar } pos(x, \Gamma)$

| $\mathcal{C}(\Gamma, \text{fun } x \Rightarrow e) =$
 branch ℓ'

ℓ : define
 $\mathcal{C}(\Gamma x, e)$
 undefine
 swap
 restore
 swap
 ubbranch

ℓ' : remember ℓ
 capture $pos(FV(e) \setminus \{x\}, \Gamma)$
 close

| $\mathcal{C}(\Gamma, c) = \dots$

| $\mathcal{C}(\Gamma, e_1 e_2) =$
 remember ℓ
 capture ζ_v
 $\mathcal{C}(\Gamma, e_2)$
 $\mathcal{C}(\Gamma, e_1)$
 open
 restore
 ubbranch

ℓ : ...

Example

$C((\text{fun } x \Rightarrow x) \ 2) =$

```
remember  $\ell''$ 
capture  $\zeta_v$ 
remember 2
branch  $\ell'$ 
 $\ell$  : define
      getvar 0
      undefine
      swap
      restore
      swap
      ubranch
 $\ell'$  : remember  $\ell$ 
      capture  $\emptyset$ 
      close
      open
      restore
      ubranch
 $\ell''$  : ...
```


Example

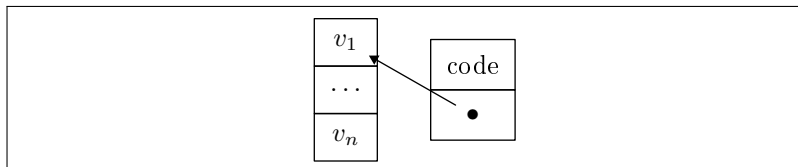
$C((\text{fun } x \Rightarrow y) \ 2) =$

```
remember  $\ell''$ 
capture  $\zeta_v$ 
remember 2
branch  $\ell'$ 
 $\ell$  : define
      getvar 0
      undefine
      swap
      restore
      swap
      ubranch
 $\ell'$  : remember  $\ell$ 
      capture  $y$ 
      close
      open
      restore
      ubranch
 $\ell''$  : ...
```

Représentation d'une fermeture

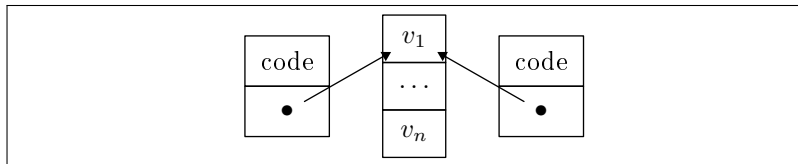
- | Du point de vue de la machine virtuelle, une fermeture est un couple formé d'un pointeur de code et d'une pile de variables capturées.
- | On peut choisir sa représentation interne : il suffit que les instructions `capture`, `close`, `open` et `restore` se comportent de façon cohérente.

Deux mots : le PC et un pointeur



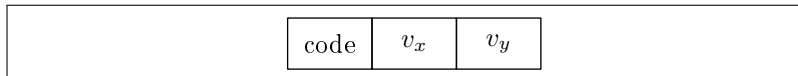
- | $\text{taille(fermeture)} = 2 * \text{taille(pointeur)} + \text{taille(environnement)}$
 - | capture : pousse un pointeur sur ζ_v au sommet de ζ_r .
 - | restore : écrase le pointeur courant de ζ_v par celui situé au sommet de ζ_r .
 - | open : copie un pointeur de code et un pointeur au sommet de la pile ζ_r .
 - | close : copie un pointeur de code et un pointeur du sommet de la pile ζ_r dans une paire placée au sommet de ζ_r .
-
- ✗ Gain en espace mémoire.
 - ✗ Impose une structure de pile persistante dont on puisse prendre l'adresse.
 - ✗ Accès lent aux variables (il faut suivre des pointeurs alloués dynamiquement).

Deux mots : le PC et un pointeur



- | $\text{taille(fermeture)} = 2 * \text{taille(pointeur)} + \text{taille(environnement)}$
 - | capture : pousse un pointeur sur ζ_v au sommet de ζ_r .
 - | restore : écrase le pointeur courant de ζ_v par celui situé au sommet de ζ_r .
 - | open : copie un pointeur de code et un pointeur au sommet de la pile ζ_r .
 - | close : copie un pointeur de code et un pointeur du sommet de la pile ζ_r dans une paire placée au sommet de ζ_r .
-
- ✗ Gain en espace mémoire.
 - ✗ Impose une structure de pile persistante dont on puisse prendre l'adresse.
 - ✗ Accès lent aux variables (il faut suivre des pointeurs alloués dynamiquement).

Un unique bloc



- | $\text{taille}(\text{fermeture}) = \text{taille}(\text{pointeur}) + \text{taille}(\text{environnement})$
 - | capture : alloue un tableau au sommet de ζ_r pour y recopier ζ_v .
 - | restore : fait pointer ζ_v vers le tableau au sommet de ζ_r .
 - | open : copie un pointeur de code et un pointeur de tableau au sommet de ζ_r .
 - | close : copie un pointeur de code et un pointeur vers un tableau de la pile ζ_r vers une paire.
-
- × Représentation coûteuse en mémoire.
 - × Accès aux variables plus efficace.

Les fonctions récursives

- | Comment compiler l'expression suivante ?

$$\text{fi } x (\text{fun } loop' \Rightarrow \text{fun } x \Rightarrow loop' (x + 1))$$

- | Quelles formes prennent les fermetures produites par l'opérateur `fi x` ?

Comment représenter les fermetures récursives ?

- | Pour compiler une valeur récursive, on commence par allouer une fermeture dans laquelle les appels récursifs sont représentés par les occurrences d'une variables libres.
- | Les appels récursifs semblent donc nécessiter un accès à l'environnement.
- | Pour construire une fermeture récursive utilisée par cette méthode, on procède en deux temps :
 1. On alloue une fermeture avec une référence par défaut associée à x dans l'environnement.
 2. On met à jour la référence à x dans l'environnement avec la véritable référence de la fermeture.

⇒ On crée un cycle.

Comment représenter les fermetures récursives ?

- | La représentation à l'aide d'un cycle introduit une **indirection** :
 - ▶ peu naturelle (un appel récursif est déjà dans la bonne fermeture) ;
 - ▶ sous-optimal (le saut à une adresse inconnue).
- | La fermeture courante est identique à celle récupérée dans l'environnement !
- | On compile les appels récursifs d'une façon spécifique pour réutiliser la fermeture courante.

Et les fonctions mutuellement récursives ?

```
...  
let rec g z = h z in  
and h t = g t in  
...
```

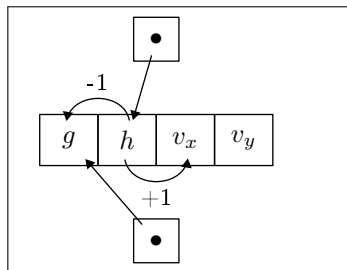
- | Moyennant l'ajout (sans difficulté) de n-uplets à notre langage, on peut écrire ce programme :

```
let m = fix (fun m => (fun z => (snd m) z, fun t => (fst m) t))  
let g = fst m  
let h = snd m  
...
```

- | On construit alors N fermetures mutuellement récursives, une pour chaque fonction.
- ⇒ En fait, on peut faire beaucoup mieux !

Compilation efficace des fonctions mutuellement récurrentes

```
let f x y =  
  let rec g z = z - h x  
  and h t = g y + t  
  in  
  ...
```



- | Il est possible de partager un unique bloc pour représenter toutes les fonctions mutuellement récurrentes :
 1. l'environnement est formé à partir de l'union des variables libres des fonctions
 2. les pointeurs de code des différentes fonctions sont agglomérés en tête du bloc.
- | Chaque fonction se voit affecter un pointeur qui peut être placé après la tête de la fermeture.
- | Pour plus de détails :
Compiling with continuations, Cambridge University Press, pages 107-108.

Synthèse

- | Les fermetures représentent du **code clos**.
 - | Elles peuvent être manipulées de façon arbitraire sans que l'on puisse rendre invalide le code qu'elles contiennent.
 - | Les langages d'ordre supérieur comme les langages fonctionnels ou à objets s'appuient d'une façon ou d'une autre sur ce mécanisme.
 - | Les langages à objets sont aussi des langages **impératifs** : les objets ont un état interne qu'ils modifient à travers les appels de méthode.
- ⇒ Ce sera le sujet du prochain cours.