Introduction à la compilation

Christine Paulin-Mohring

Université Paris Sud

Master Informatique 2008-2009

Sommaire

Objectifs du cours

Principes de construction d'un compilateur

Programme: description d'une suite d'opérations qui étant donnés une entrée va produire un résultat
Plusieurs modèles de calcul possible (impératif, fonctionnel . . .)

Autres apports

- maitriser les techniques de base pour la transformation textuelle (lex, yacc)
- comprendre les caractéristiques des langages de programmation (sémantique, efficacité)
- apprendre à programmer: structures de données et algorithmes avancés

Organisation

- TDs à partir d'un recueil d'exercices
  début semaine 15 septembre
- Projet associé (responsable Sylvain Conchon)
  début semaine 15 septembre
- Utilisation du langage ocaml
  Mise à niveau jeudi 11 septembre
- Partiel + Examen

Calendrier

10/09 Cours 1 : Introduction aux compilateurs

11/09 Cours 2 : Analyse lexicale et syntaxique 1

15/09 Cours 3 : Analyse lexicale et syntaxique 2
TD 1 - Projet 1

22/09 Cours 4 : Analyse sémantique 1
TD 2 - Projet 2

29/09 Cours 5 : Analyse sémantique 2
TD 3 - Projet 3

06/10 Cours 6 : Analyse sémantique 3
TD 4 - Projet 4

13/10 Cours 7 : Génération de code 1
TD 5 - Projet 5

20/10 Cours 8 : Génération de code 2
TD 6 - Projet 6

27/10 Partiel

Calendrier-suite

03/11	Cours 9 : Génération de code 3 TD 7 - Projet 7
10/11	Cours 10 : Génération de code 4 TD 8 - Projet 8
17/11	Cours 11 : Organisation de la mémoire TD 9 - Projet 9
24/11	TD 10 - Projet 10
01/12	TD 11 - Projet 11
??	Projet 12-soutenances

Pratique

Page WEB du cours :

`http://www.lri.fr/~paulin/COMPIL`

Coordonnées :

Adresse électronique	`Christine.Paulin@lri.fr`
Téléphone	01 72 92 59 05
Bureau	INRIA Saclay - Île-de-France, Parc Orsay Université, Bat N, 1er étage.

Merci d'utiliser prioritairement le courrier électronique.

Prérequis

- Théorie des langages formels: expressions régulières, automates, grammaires, ambiguité
- Langages de programmation: typage, organisation de la mémoire
- Algorithmique : tables de hachage, arbres, graphes. . .

Bibliographie

Compilation

- A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998
- R. Wilhelm and D. Maurer. *Les compilateurs: théorie, construction, génération*. Manuels Informatique. Masson, 1994
- J. R. Levine, T. Mason, and D. Brown. *Lex & Yacc*. Unix Programming Tools. O' Reilly, 1995

ocaml

- E. Chailloux, P. Manoury, and B. Pagano. *Développement d'applications avec Objective Caml*. O'Reilly, 2000
- Claude Marché and Ralf Treinen. Formation au langage CAML. Université Paris Sud. Notes du Cours LGL
- Jean-Christophe Filliâtre. Intiation à la programmation fonctionnelle. Université Paris Sud. Notes de Cours - Master Informatique M1

Plan-1

Introduction à la compilation

- Structure d'un compilateur, exemple
- Evaluateur versus compilateur
- Techniques de construction de compilateurs

Analyse lexicale et syntaxique

- Mise en oeuvre d'un analyseur lexical
- Analyse descendante, analyse ascendante
- Ambiguités et précédences
- Actions sémantiques: arbres de syntaxe abstraite

Plan-2

Analyse sémantique

- Analyse de portée
- Typage : surcharge, inférence de types
- Analyses statiques

Génération de code

- Machines abstraites
- Appels de fonctions, récursion terminale
- Allocations de registres
- Gestion de la mémoire

Sommaire

Sommaire

Qu'est-ce qu'un compilateur ?

- Un compilateur est un traducteur qui permet de transformer un programme écrit dans un langage L_1 en un autre programme écrit dans un langage machine L_2.
- En pratique on s'arrête souvent à un langage intermédiaire (assembleur ou encore langage d'une machine abstraite).

Qu'est-ce qu'un programme ?

- Description de comment à partir d'une entrée, produire un résultat.
- L'exécution du programme peut ne pas terminer ou échouer à produire un résultat.

Qu'attend-on d'un compilateur ?

Détection des erreurs :

- Identificateurs mal formés, commentaires non fermés ...
- Constructions syntaxiques incorrectes
- Identificateurs non déclarés
- Expressions mal typées `if 3 then "toto" else 4.5`
- Références non instanciées
- ...

Les erreurs détectées à la compilation s'appellent les erreurs statiques.

Les erreurs détectées à l'exécution s'appellent les erreurs dynamiques: division par zéro ou dépassement des bornes dans un tableau.

Qu'attend-on d'un compilateur ? (2)

- L'efficacité
  - Le compilateur doit être si possible rapide (en particulier ne pas boucler)
  - Le compilateur doit produire un code qui s'exécutera aussi rapidement que possible

- La correction: le programme compilé doit représenter le même calcul que le programme original.

Structure d'un compilateur

Un compilateur se construit en plusieurs phases.

- Analyse syntaxique :
  - Transforme une suite de caractères en un arbre de syntaxe abstraite (ast) représentant la description des opérations à effectuer.
  - Combinaison de l'analyse lexicale qui reconnait des "mots" aussi appelés token ou entités et de l'analyse syntaxique qui reconnait des phrases bien formées.

- Analyse sémantique : analyse l'arbre de syntaxe abstraite pour calculer de nouvelles informations permettant de:
  - rejeter des programmes incorrects (portée, typage. . .)
  - préparer la phase de génération de code (organisation de l'environnement, construction de tables pour les symboles, résolution de la surcharge . . .)

Structure d'un compilateur (2)

- Génération de code : passage par plusieurs langages intermédiaires pour produire un code efficace
  - Organisation des appels de fonctions dans des tableaux d'activation
  - Analyses de flots
  - Allocation de registres
  - ...

Sommaire

Exemple

- Compiler un langage ARITH d'expressions arithmétiques vers le code d'une machine à pile.

Le langage ARITH se compose d'une suite d'expressions:

```
set ident = expr
print expr
```

Les expressions peuvent comporter des variables locales.

Exemple de programme:

```
set x = 4
set xx = let y = 2 * x + 5 in - (y * y)
```

Description syntaxique du langage

<inst> ::= **set** <ident> = <expr>
| **print** <expr>
<expr> ::= <entier> | <ident> |
| <expr> <binop> <expr> | <unop> <expr>
| **let** *ident* = <expr> **in** <expr>
| (<expr>)
<binop> ::= + | - | * | /
<unop> ::= -

Les éléments à préciser

- Qu'est-ce qu'un identificateur ?
  (alpha-numérique, commence par une lettre)
- Qu'est-ce qu'un entier ?
  (suite de chiffres)
- Quels sont les mots clés ?
  (non utilisés comme identificateur)
- Comment se lèvent les ambiguités
  (règles usuelles pour les expressions arithmétiques)

Choix de la structure d'arbre de syntaxe abstraite

- Doit représenter l'information contenue dans le programme et qui est utile à l'analyse sémantique et la génération de code.
- Construction *simple* à partir de la grammaire.

Quelques exemples:

- Les parenthèses sont utiles pour résoudre les ambiguités d'une écriture linéaire mais inutiles dans les arbres de syntaxe abstraites.
- Les commentaires sont inutiles pour engendrer le code.
- La localisation des constructions de programmes dans le fichier permet de mieux rendre compte des erreurs sémantiques.

Structure des ast pour ARITH

```
type prg = instr list
and instr = Set of string*expr | Print of expr
and expr =
    Cst of int
  | Var of string
  | Op of binop*expr*expr
  | Letin of string*expr*expr
and binop = Sum | Diff | Prod | Quot
;;
```

L'analyseur lexical

Principes

- Utilise des expressions régulières pour reconnaître des entités lexicales (tokens) qui sont fournies à la grammaire (symboles terminaux).
- Ces entités doivent être déclarées
- Certaines entités (identificateurs, constantes entières...) portent des valeurs qui seront utilisées pour construire l'ast.
- Le type ocaml des tokens

```
type token =
    CST of int | IDENT of string
  | SET | LET | IN | PRINT | EOF | LP | RP
  | PLUS | MINUS | TIMES | DIV | EQ
```

Exemple de fichier ocamllex : `lexer.mll`

Quelques abbréviations pour les expressions régulières:

```
let letter = ['a'-'z' 'A'-'Z']
let digit = ['0'-'9']
let ident = letter (letter | digit)*
let integer = ['0'-'9']+
let space = [' ' '\t' '\n']
```

Les règles de reconnaissance

```
rule nexttoken = parse
   | '+'      { PLUS }
   | '-'      { MINUS }
   | '*'      { TIMES }
   | '/'      { DIV }
   | '='      { EQ }
   | '('      { LP }
   | ')'      { RP }
   | integer { CST (int_of_string (lexeme lexbuf)) }
```

Cas particuliers

Traitement des espaces, des mot-clés, fin de fichier et erreurs

```
rule nexttoken = parse
  | ...
  | space+   { nexttoken lexbuf }
  | ident    { id_or_kwd (lexeme lexbuf) }
  | eof      { EOF }
  | _        { raise (Lexing_error (lexeme lexbuf)) }
```

Entete de fichier

Déclare les fonctions utiles à l'opération d'analyse

```
{
open Lexing
open Parser
exception Lexing_error of string

let kwd_tbl =
 ["let",LET;"in",IN; "set",SET;"print",PRINT]
let id_or_kwd s =
 try List.assoc s kwd_tbl with _ -> IDENT(s)
}
```

Les outils de type lex

La commande

```
ocamllex lexer.mll
```

produit un fichier ml `lexer.ml` qui lui-même contient une fonction

```
val nexttoken : Lexing.lexbuf -> Parser.token
```

- Le module `Lexing` d'ocaml introduit des fonctions de manipulation lexicales.
- Des fonctions comme `lexeme` permettent d'accéder à la chaîne de caractères correspondant à l'expression régulière reconnue.
- Dans cet exemple, le type des tokens est défini dans le fichier de grammaire `Parser`.
- Un objet de type `Lexing.lexbuf` (buffer pour l'analyse lexicale, comportant des informations sur le positionnement dans le fichier) peut être construit à partir d'un fichier ou d'une chaîne de caractères.

Grammaire

- Une grammaire se définit par des symboles terminaux, non-terminaux et des règles de dérivation.
- Un *mot* est reconnu si on peut construire un arbre de dérivation syntaxique dont les feuilles sont les terminaux et les noeuds correspondent aux règles.
- Les tokens fournis par l'analyseur lexical correspondent aux terminaux de la grammaire.
- Les règles suivent les constructions syntaxiques du langage mais il faut tenir compte des ambiguités.
- Les actions associées aux règles permettent de construire les ast.

Exemple de fichier ocamlyacc

Fichier `parser.mly`
Le cœur du programme:

```
instr:
| SET IDENT EQ expr { Set($2,$4)}
| PRINT expr       { Print($2) }
;

expr:
| CST                          { Cst($1) }
| IDENT                        { Var($1) }
| expr PLUS expr               { Op(Sum,$1,$3) }
| expr MINUS expr              { Op(Diff,$1,$3) }
| expr TIMES expr              { Op(Prod,$1,$3) }
| expr DIV expr                { Op(Quot,$1,$3) }
| MINUS expr %prec uminus      { Op(Diff,Cst(0),$2) }
| LET IDENT EQ expr IN expr    { Letin($2,$4,$6) }
| LP expr RP                   { $2 }
;
```

Préambule

- Déclaration des tokens
- Précision des précédences

```
%token <int> CST
%token <string> IDENT
%token SET, LET, IN, PRINT
%token EOF
%token LP RP
%token PLUS MINUS TIMES DIV
%token EQ

/* priorités et associativités des tokens */
%nonassoc IN
%left MINUS PLUS
%left TIMES DIV
%nonassoc uminus
```

Points d'entrée

```
/* Point d'entrée de la grammaire */
%start prog

/* Type des valeurs retournées par l'analyseur syntaxique *
%type <Ast.prg> prog

%%
prog:
| instrs EOF {List.rev $1}
;
instrs:
| instr         { [$1] }
| instrs instr { $2::$1 }
;
```

Les outils de type Yacc

La commande

```
ocamlyacc parser.mly
```

produit un fichier ml `parser.ml` qui contient la déclaration du type des tokens ainsi que la fonction d'analyse:

```
val prog :
  (Lexing.lexbuf -> token) -> Lexing.lexbuf -> Ast.prg
```

Pour construire un ast correspondant à un fichier `file`

```
let f = open_in file in
let buf = Lexing.from_channel f in
Parser.prog Lexer.nexttoken buf
```

A propos des outils d'analyse

- Les outils comme ocamllex et ocamlyacc font appel à des algorithmes complexes de calculs d'automates:
  - analyse lexicale : automates finis
  - analyse syntaxique : automates à pile

- Nous étudierons le fonctionnement de ces outils sans entrer dans le détail du calcul des tables.

Sommaire

Sémantique des langages

- La sémantique sert à préciser la nature des calculs représentés par un programme
- La plupart des langages courants ne disposent pas d'une sémantique formelle complète (les compilateurs peuvent donc faire des choix multiples).
- Il y une grande diversité de sémantiques pour les langages de programmation (dénotationnelle, axiomatique, statique, . . .).

Sémantique opérationnelle à grand pas

- Elle explique la valeur finale calculée par un programme en fonction de l'environnement dans lequel il est exécuté.
- Elle s'appuie sur un modèle des valeurs sémantiques.
- Elle s'exprime sous forme de règles d'inférences qui permettent de définir la relation d'évaluation.

Exemple

Cas du langage ARITH:

- Les valeurs manipulées pour les expressions sont uniquement des entiers.
- Dans le langage de programmation, l'opération $+$ s'applique à deux expressions quelconques.
  Dans le modèle sémantique, on a des opérations pour effectuer les expressions correspondantes sur les entiers.
- Un environnement relie des valeurs à des variables.

Les règles expliquent comment chaque programme produit une valeur:

- dans un environnement donné, une expression s'évalue vers un entier (si toutes les variables mentionnées sont définies dans l'environnement)
- un programme s'évalue également dans un environnement (initialement vide) et a pour valeur une liste des valeurs imprimées.

Exemple de règles

$$\frac{\rho \vdash e_1 \rightsquigarrow n_1 \quad \rho \vdash e_2 \rightsquigarrow n_2}{\rho \vdash \texttt{Op}(\texttt{PLUS}, e_1, e_2) \rightsquigarrow n_1 + n_2}$$

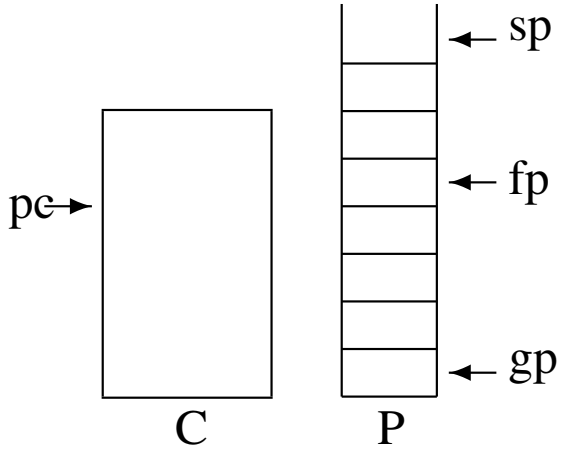
$$\frac{\rho \vdash e_1 \rightsquigarrow n_1 \quad \rho + (x, n_1) \vdash e_2 \rightsquigarrow n_2}{\rho \vdash \texttt{Letin}(x, e_1, e_2) \rightsquigarrow n_2}$$

$$\frac{\rho \vdash e \rightsquigarrow n \quad \rho + (x, n) \vdash p \rightsquigarrow l}{\rho \vdash (\texttt{Set}(x, e) :: p) \rightsquigarrow l}$$

$$\frac{\rho \vdash e \rightsquigarrow n \quad \rho \vdash p \rightsquigarrow l}{\rho \vdash (\texttt{Print}(e) :: p) \rightsquigarrow (n :: l)}$$

Une machine virtuelle à pile

L'organisation de la machine virtuelle est décrite par la figure suivante:



- zone de code `C`
- registre *pc* qui pointe sur l'instruction courante à exécuter.
- pile `P` permettant de stocker des valeurs entières.
- Trois registres permettent d'accéder à différentes parties de `P`:
  - *sp* (stack pointer) pointe sur le premier emplacement vide de la pile,
  - *fp* (frame pointer) repère l'adresse de base des variables locales
  - *gp* pointe sur la base de la pile à partir de laquelle sont stockées les variables globales.

Sémantique de la machine virtuelle

- la valeur correspond à la configuration de la machine
- les programmes sont des listes d'instructions
- on donne une sémantique opérationnelle à petit pas qui décrit la transformation de l'état lors de chaque instruction.

Règles sémantiques

Code	Description	*sp*	Condition
`ADD`	`P[sp-2]:=P[sp-2]+P[sp-1]`	`sp-1`	`P[sp-2]`, `P[sp-1]` entiers
`SUB`	`P[sp-2]:=P[sp-2]-P[sp-1]`	`sp-1`	`P[sp-2]`, `P[sp-1]` entiers
`MUL`	`P[sp-2]:=P[sp-2]*P[sp-1]`	`sp-1`	`P[sp-2]`, `P[sp-1]` entiers
`DIV`	`P[sp-2]:=P[sp-2]/P[sp-1]`	`sp-1`	`P[sp-2]`, `P[sp-1]` entiers
`PUSHI n`	`P[sp]:=n`	`sp+1`	`n` entier
`PUSHN n`	`P[sp]:=0 ...P[sp+n-1]:=0`	`sp+n`	`n` entier
`PUSHG n`	`P[sp]:=P[gp+n]`	`sp+1`	`n` entier
`STOREG n`	`P[gp+n]:=P[sp-1]`	`sp-1`	`n` entier
`PUSHL n`	`P[sp]:=P[fp+n]`	`sp+1`	`n` entier
`STOREL n`	`P[fp+n]:=P[sp-1]`	`sp-1`	`n` entier
`WRITEI`	imprime `P[sp-1]` sur l'écran	`sp-1`	`P[sp-1]` entier
`START`	Affecte la valeur de `sp` à `fp`	`sp`	
`STOP`	Arrête l'exécution du programme	`sp`	

Schéma de compilation

Avant d'écrire la fonction de compilation, il faut déterminer la correspondance entre les entités des deux langages.

- Le langage ARITH manipule des noms de variables
- La machine virtuelle stocke les données dans la pile et les repère par un décalage par rapport à `fp` ou `gp`.
- Chaque variable *x* de l'environnement est stockée à un endroit précis (local par rapport à `fp` ou global par rapport à `gp`) dans la pile.
  Le choix d'affectation est fait à l'analyse sémantique et conservé dans une table des symboles.

Schéma de compilation (2)

- Invariant à déterminer: où se retrouve la valeur du programme?
  - Valeur en sommet de pile
  - Correspondance entre les valeurs dans la pile et l'environnement
- Formulation mathématique:

$$\frac{\rho \vdash e \rightsquigarrow n \quad P \leftrightarrow_T \rho \quad \texttt{compile}(T, e) = lc \quad (P, sp) + lc \rightsquigarrow (P', sp')}{P'[sp' - 1] = n}$$

- On garantit également que l'environnement n'a pas été altéré:

$$\forall x, P[T(x)] = P'[T(x)]$$

Sommaire

Interpréteur

- On peut évaluer les programmes d'un langage L en utilisant un autre langage de programmation dès que l'on connait les entrées du programme.
- Implantation plus ou moins sophistiquée des règles de sémantique.
- L'interpréteur ajoute un niveau supplémentaire d'exécution qui le rend en général moins efficace que le compilateur mais il est aussi plus facile à écrire (on peut utiliser toute la puissance du langage de programmation).
- L'interpréteur présuppose la connaissance des entrées du programmes, il ne permet pas de modulariser la construction des résultats.
- Le compilateur a juste besoin de savoir où il pourra trouver les valeurs au moment de l'exécution.
- Il n'est pas forcément nécessaire de connaître précisément les valeurs des entrées pour faire des évaluations intéressantes. On peut également calculer avec des valeurs abstraites comme les types.

Sommaire

Techniques de construction de compilateurs

- De nombreux compilateurs (Java, ocaml) sont écrits dans le langage lui-même
- On utilise un mécanisme d'auto-compilation (boot-strap)
  - un compilateur ou évaluateur élémentaire écrit dans un langage de bas niveau
  - des compilateurs de plus en plus avancés écrits dans le langage de haut niveau
- On peut également faire de la compilation croisée qui permet de créer des compilateurs pour des architectures multiples.

Voir exercices de TD

Analyse lexicale et syntaxique

Rappel

Objectif de l'analyse lexicale et syntaxique
Entrée : suite de caractères
Sortie : arbre de syntaxe abstraite représentant le programme sous-jacent
Outils théoriques : expressions régulières, grammaires
Outils logiciels : Lex et Yacc

Sommaire

Sommaire

Principes de l'analyse lexicale

- Chaque entité lexicale est décrite par une expression régulière.
- A chaque entité lexicale est associée une action.
- L'action produit en général un token, mais peut aussi avoir d'autres effets (compter des lignes, transformer le texte).

Sommaire

Expressions régulières

Les constructions de base:

$$
\begin{array}{rcl}
\textit{regexp} & ::= & \textit{'caractère'} \\
& | & \textit{regexp regexp} \\
& | & \textit{regexp} \mid \textit{regexp} \\
& | & \textit{regexp} \star
\end{array}
$$

Dans la théorie, on ajoute aussi des expressions régulières pour:

- le mot vide notée ϵ
- l'ensemble vide notée $\emptyset$

mais qui ne sont pas utilisées dans les analyseurs lexicaux. Chaque expression e représente un langage (ensemble de mots $L(e)$) qui est dit régulier.

$$
\begin{array}{ll}
L(c) & = \{c\} \\
L(e_1\ e_2) & = \{m_1 m_2 | m_1 \in L(e_1), m_2 \in L(e_2)\} \\
L(e_1|e_2) & = L(e_1) \cup L(e_2) \\
L(e^*) & = \bigcup_{0 \leq n} L(e^n) \qquad e^0 = \epsilon \;\; e^{n+1} = e(e^n)
\end{array}
$$

Expressions régulières étendues

La syntaxe autorisée par ocamllex:

regexp	::=	"*chaîne*"
	|	*regexp* `?`
	|	*regexp* `+`
	|	`[` *caractères* `]`
	|	`[^` *caractères* `]`
	|	`eof`
	|	`_`
	|	`(` *regexp* `)`
	|	*regexp* `as` *ident*
caractères	::=	(`'`*caractère*`'` | `'`*caractère*`'` `-` `'`*caractère*`'`)`+`

Précédence plus haute `+` et `*`, ensuite `?`, puis la concaténation puis le choix `|`.
Exemple: `'a'|'b'* 'c'+`.

Résultats

- Si e est une expression régulière alors il existe un automate fini (que l'on calcule à partir de e) qui reconnait le langage $L(e)$.
- La réciproque est vraie.
- Le langage des expressions bien parenthésées $\{a^n b^n | n \in \mathbb{N}\}$ n'est pas régulier.

Exercice donner une expression régulière pour les identificateurs (suite de caractères alphabétiques) qui ne sont pas le mot clé **set**.
Construire l'automate correspondant.

Sommaire

Syntaxe d'un fichier ocamllex

```
  { header }
let ident = regexp ...
rule entrypoint [arg1... argn] =
  parse regexp { action }
      | ...
      | regexp { action }
and entrypoint [arg1... argn] =
  parse ...
and ...
{ trailer }
```

Résultat de ocamllex

La commande

`ocamllex` *`file.mll`*

produit un fichier *file.ml* qui doit ensuite être compilé par ocaml.
affiche la taille de l'automate et de la table stockant les transitions.

- Chaque *entrypoint* définit une fonction ocaml.
- Ces fonctions peuvent être paramétrées et ont toujours un argument supplémentaire `lexbuf`.
- Les fonctions peuvent être appelées récursivement dans les actions.

Principe de fonctionnement de ocamllex

Le fichier ocaml engendré:

```
 Le code ocaml  du prélude
let lex_tables = { ... }
let rec entrypoint lexbuf =
    lex_entrypoint_rec lexbuf 0
and lex_entrypoint_rec lexbuf lex_state =
  match Lexing.engine lex_tables lex_state lexbuf with
    | 0 ->
        ( code de l'action de l'expr reconnue à l'état 0 )
    | 1 ->
        ( code de l'action de l'expr reconnue à l'état 1 )
    ...
    | lex_state -> lexbuf.Lexing.refill_buff lexbuf;
                       lex_entrypoint_rec lexbuf lex_state
 Le code ocaml  du postlude
```

Principe de fonctionnement de ocamllex (2)

- Des erreurs dans le code ocaml du prélude, des actions ou du postlude ne seront repérées que lors de la compilation de *file.ml* par ocaml.
- Des directives dans le fichier *file.ml* `#18 "test.mll"` permettent de relier les lignes de *file.ml* à celles du fichier ocamllex correspondant.
- La table de transitions associe à un état et un caractère l'état suivant dans l'automate.
- Le stockage de la table de transitions utilise des techniques de représentation de matrices creuses de manière linéaire dans des chaines de caractères.

Un peu d'algorithmique

On veut représenter une matrice $M(i, j)$ de taille $n \times m$.

On suppose que de nombreuses cases de M ont une valeur constante d et on cherche à représenter M en évitant d'écrire d.

- On utilise un tableau T et on cherche à placer les lignes de M sans les valeurs d dans le tableau T.
- Pour savoir à quelle ligne appartient un élément de T, on lui associe un second tableau C de même taille.

Un peu d'algorithmique (2)

- Pour placer la $i$ème ligne dans T:
  - on cherche une position p la plus petite possible telle que si $M(i,j) \neq d$ alors $T(p+j)$ est vide.
  - On stocke p dans un tableau $d[i]$ de taille n.
  - Pour tout j tel que $M(i,j) \neq d$ on met à jour $T[p+j] := M(i,j)$ et $C[p+j] := i$
- La fonction d'accès à $M(i,j)$ à partir de T, d et C sera vue en TD. Cette fonction a un coût constant.
- On peut choisir une valeur par défaut différente pour chaque ligne et la conserver dans une autre table.

Commentaires

Exercice

- Trouver l'expression régulière correspondant à des commentaires qui débutent par $/*$ et se terminent à la première occurrence de $*/$
- Construire l'automate correspondant
- Peut-on mettre entre commentaire un code qui contient des commentaires ?
- Quel est l'avantage des commentaires qui débutent par un caractère spécial par exemple $\#$ et se terminent par la fin de ligne ?

Commentaires imbriqués

Les analyseurs lexicaux permettent de traiter des commentaires imbriqués:

- Possibilité d'avoir un compteur pour marquer le nombre de commentaires restant à fermer.

```
rule nexttoken = parse
   "/*"    {com:=1; comment lexbuf; nexttoken lexbuf}
and comment = parse
   "*/"    {if !com > 1 then
               begin decr com; comment lexbuf end}
 | "/*"    {incr com; comment lexbuf}
 | eof     {raise Error "commentaire non terminé"}
 | _       {comment lexbuf}
```

Commentaires imbriqués (2)

- Utilisation de la pile d'appels

```
  rule nexttoken = parse
   "/*"   {comment lexbuf; nexttoken lexbuf}
and comment = parse
   "*/"   {}
 | "/*"   {comment lexbuf; comment lexbuf}
 | eof    {raise Error "commentaire non terminé"}
 | _      {comment lexbuf}
```

Le cas des chaînes de caractères

- Les expressions régulières pour les chaînes de caractère sont assez complexes.
  Par exemple on écrira `"Nom:\"PP\"\\\n"` pour représenter la chaîne `Nom:"PP"\` terminée par un retour chariot.
- Elles peuvent aussi être arbitrairement longues.
- Il est recommandé de lire les chaînes avec une entrée de l'analyseur qui met les caractères au fur et à mesure dans un buffer.

Sommaire

Localisation des erreurs

- ocamllex conserve dans la variable `lexbuf` le nombre de caractères par rapport au début du fichier
  - `Lexing.lexeme_start lexbuf` de type `int` le nombre de caractères jusqu'au début de l'entité reconnue.
  - `Lexing.lexeme_end lexbuf` le nombre de caractères jusqu'à la fin de l'entité reconnue.
- Un type position défini dans `Lexing`.

```
type position =
  pos_fname : string;   (* nom fichier *)
  pos_lnum : int;       (* no de ligne *)
  pos_bol : int;        (* nbre de car entre début de fichier
                           et début de ligne *)
  pos_cnum : int;       (* nbre de car entre début de fichier
                           et position courante *)
```

- Accès à la position: `lexbuf.Lexing.lex_curr_p` de type `position`
  Attention seul `pos_cnum` est mis à jour automatiquement.

Mise à jour des numéros de ligne

- En prélude de l'analyseur une fonction à appeller à chaque retour chariot:

```
let newline lexbuf =
let pos = lexbuf.lex_curr_p in
lexbuf.lex_curr_p <-
 { pos with pos_lnum = pos.pos_lnum + 1;
            pos_bol = pos.pos_cnum }
```

- Dans l'analyseur:

```
rule nexttoken = parse
      | '\n'     { newline lexbuf; nexttoken lexbuf }
```

- A faire aussi dans les commentaires ...

Autres utilisations des analyseurs lexicaux

Les analyseurs lexicaux sont aussi utiles pour d'autres tâches que la reconnaissance de tokens. Par exemple pour faire un préprocesseur qui définit des constantes et des directives de compilation (qui peuvent être imbriqués).

```
#define nom valeur
#define A
#ifdef A
dans le cas où A est définie
#else
dans le cas contraire
#endif
```

Les directives sont sur une ligne.

Modélisation du pré-processeur

- Les définitions de macros sont dans une table de hachage:

```
let definitions = Hashtbl.create 97
let define = Hashtbl.add definitions
let defined = Hashtbl.mem definitions
let definition = Hashtbl.find definitions
```

- Une entrée `scan` qui écrit le fichier processé.
- Une entrée `skip` qui va ignorer le texte jusqu'au `#else` ou `#endif` correspondant.
- Une entrée `skip_to_endif` qui va ignorer le texte jusqu'au `#endif` fermant.

Pré-processeur-scan

```
rule scan = parse
  | "#define" space+ (ident as x) space* ([^'\n']* as v) '\n'
      { define x v; scan lexbuf }
  | "#ifdef" space+ (ident as x) space* '\n'
      { if defined x then scan lexbuf else skip lexbuf }
  | "#else" space* '\n'
      { skip lexbuf }
  | "#endif" space* '\n'
      { scan lexbuf }
  | ident as x
      { printf "%s" (if defined x then definition x else x);
        scan lexbuf }
  | _ as c
      { printf "%c" c; scan lexbuf }
  | eof
      { () }
```

Il faudrait traiter spécifiquement les chaînes de caractères et les commentaires.

Pré-processeur-skip

```
and skip = parse
  | ("#else" | "#endif") space* '\n'
      { scan lexbuf }
  | "#ifdef" space+ ident space* '\n'
      { skip_to_endif lexbuf; skip lexbuf }
  | _   { skip lexbuf }
  | eof { () }

and skip_to_endif = parse
  | "#endif" space* '\n'
      { () }
  | "#ifdef" space+ ident space* '\n'
      { skip_to_endif lexbuf; skip_to_endif lexbuf }
  | _   { skip_to_endif lexbuf }
  | eof { () }
```

Récapitulatif

- Bien comprendre les conventions lexicales : quelle expression régulière ?
- Définir les unités lexicales : est-ce que la grammaire a besoin de distinguer deux classes ?
- Quelques “trucs”:
  - traitement des commentaires
  - traitement des mots clés
  - gestion des numéros de ligne

Sommaire

Sommaire

Rappels

Une grammaire G est composée de:

- Ensemble de terminaux: T
- Ensemble de non-terminaux: N, symbole initial $S \in N$.
- Ensemble de règles de la forme

$$X ::= x_1 \dots x_n \quad \text{avec} \quad x_i \in N \cup T$$

Exemple

$T = \{a, b\}$, $N = \{X\}$

X	::= a X b
X	::= ϵ
X	::= X X

Remarques

- Règles récursives
- Règle pour produire le mot vide

Dérivations

$$X \rightarrow XX \rightarrow aXbX \rightarrow aaXbbX \rightarrow aaXbbaXb \rightarrow aabbaXb \rightarrow aabbab$$

Langage reconnu

Une grammaire reconnait un ensemble de mots formés sur les symboles terminaux.
Ce sont les mots obtenus par dérivation à partir du symbole initial.

$$S \rightarrow m_1 \rightarrow \cdots m_n \in T^*$$

On a $m \rightarrow m'$ si:

- m contient un non-terminal X
- La grammaire contient une règle $X ::= x_1 \ldots x_n$
- m' est obtenu en remplaçant une occurrence de X par $x_1 \ldots x_n$

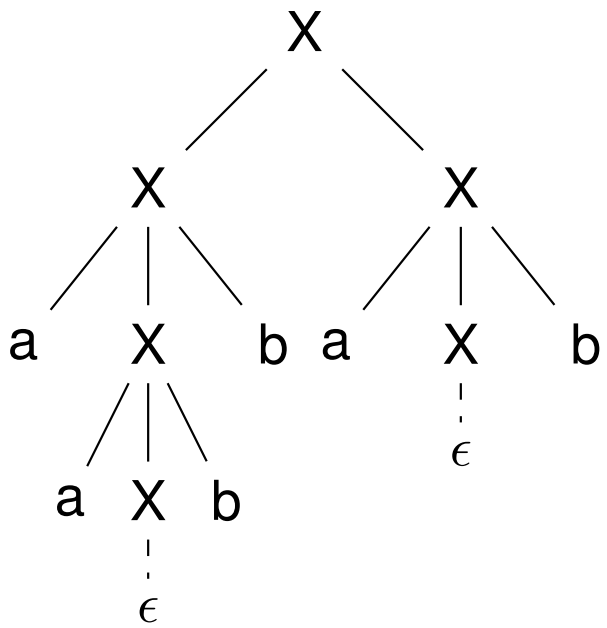
Arbre de dérivation syntaxique

Un mot m est reconnu s'il existe un arbre de dérivation syntaxique.

- les noeuds internes contiennent des symboles non-terminaux
- les feuilles contiennent des symboles terminaux
- la racine est le symbole initial S
- dans un parcours infixe, les feuilles forment le mot m
- si un noeud interne étiqueté X a des sous-arbres $t_1, \ldots, t_n$ de racines $x_1, \ldots, x_n$ alors $X ::= x_1 \ldots x_n$ est une règle de la grammaire.

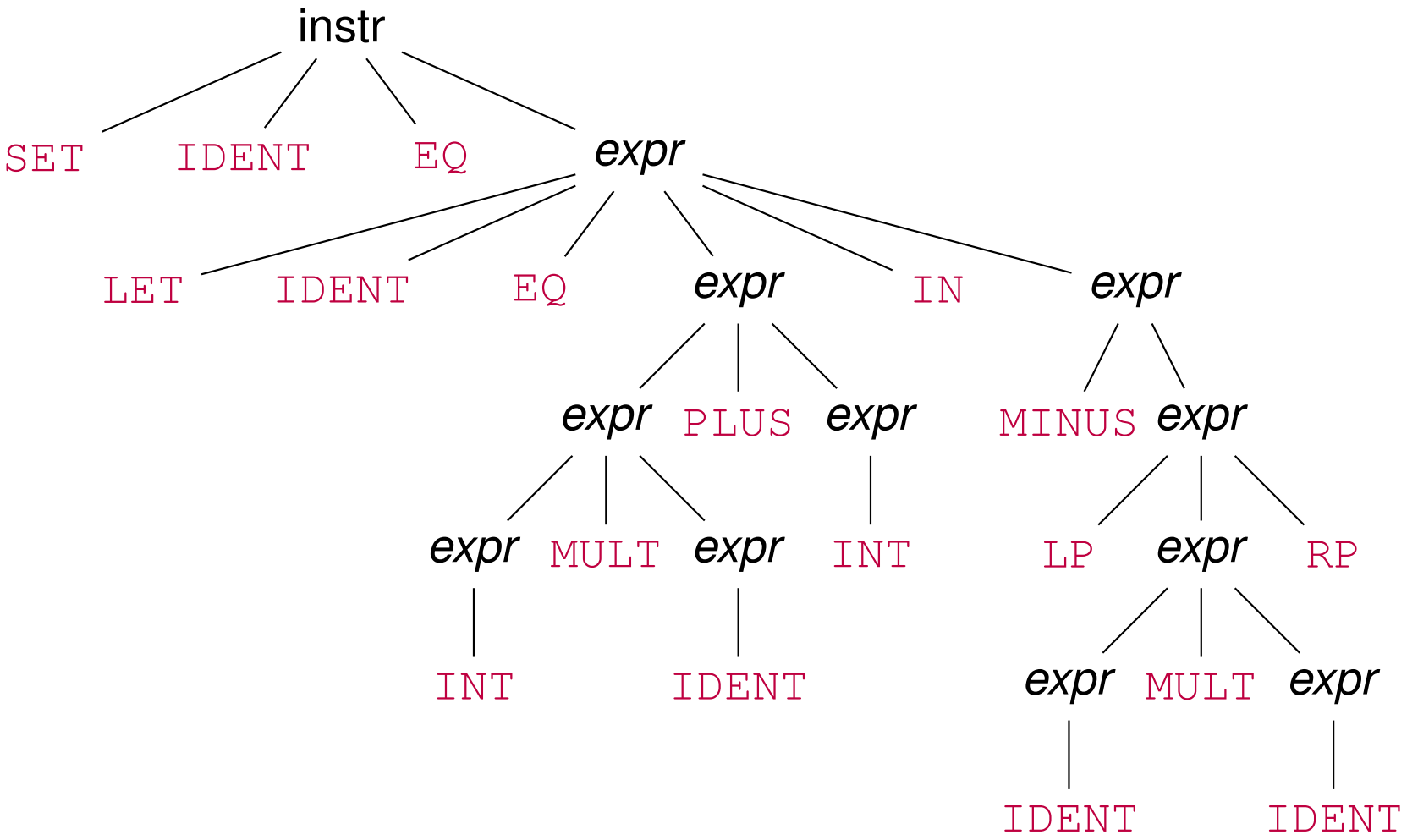
remarque
ne pas confondre arbre de dérivation syntaxique (associé à la grammaire) et arbre de syntaxe abstraite associé au langage.

Exemple



Exemple: langage ARITH

```
set xx = let y = 2 * x + 5 in - (y * y)
```



Principe de l'analyse syntaxique

- Entrée : une suite de tokens (le mot à reconnaitre)
- Recherche d'un arbre de dérivation syntaxique pour ce mot
- Reconstruction d'informations à partir de cet arbre (une action associée à chaque règle)

Comment reconstruire l'arbre ?

Sommaire

Principe de l'analyse descendante

On essaie de construire l'arbre de dérivation syntaxique en partant de la racine.

- Une fonction associée à chaque non terminal X.
- Entrée: un mot m (ou l'indice dans une chaîne globale).
- Construit un arbre de racine X dont les feuilles forment un préfixe de m.
- Renvoie l'information associée au sous-arbre de racine X et le reste du mot m.
- Suppose qu'étant donné le non-terminal et le mot, on peut décider de la règle à appliquer.
- Pour reconnaître le mot entier, on part de la fonction associée au symbole initial S, on vérifie qu'on a atteint la fin de chaîne.

Exemple

On suppose le mot *m* à reconnaître déclaré globalement et *n* représente la première lettre non traitée

```
let a n = if m.[n]='a' then (n+1,info_a())
                       else raise ParseError
let b n = if m.[n]='b' then (n+1,info_b())
                       else raise ParseError
let rec X n = match guess n with
     "aXb" -> let (n1,i1) = a n
                  in  let (n2,i2) = X n1
                  in  let (n3,i3) = b n2
                  in (n3,info_X_aXb(i1,i2,i3))
   | "ϵ" -> (n,info_X_ϵ())
   | "XX"  -> let (n1,i1) = X n
                  in  let (n2,i2) = X n1
                   in (n2,info_X_XX(i1,i2))
```

Limites de l'analyse descendante

- On ne peut pas toujours décider de la règle à appliquer.
- On s'intéresse aux grammaires LL(1) dans lesquelles on peut décider en fonction du premier caractère à lire $m.[n]$.
- Les grammaires récursives gauches ($X ::= Xm$) ne sont pas LL(1).
- On peut parfois les transformer pour obtenir des grammaires LL(1).

Exemple d'élimination de la récursion

Grammaire des expressions arithmétiques sous une forme LL(1)

S	::=	E
E	::=	T E'
E'	::=	+ T E'
E'	::=	ϵ
T	::=	F T'
T'	::=	* F T'
T'	::=	ϵ
F	::=	IDENT
F	::=	INT
F	::=	(E)

Analyseur descendant

analyseur (sans reconstruction de valeur)

```
let rec S n = let n1 = E n
              in if n <> String.length m then parseerror()
and E n = let n1 = T n in E' n1
and E' n = if m.[n]='+' then let n1 = T (n+1) in E' n1
           else n
and T n = let n1 = F n in T' n1
and T' n = if m.[n]='*' then let n1 = F (n+1) in T' n1
           else n
and F n =  if m.[n]=IDENT || m.[n]=INT then (n+1)
           else if m.[n]='(' then
                let n1 = E (n+1) in
                  if m.[n1]=')' then n1+1 else parseerror()
```

A propos d'analyse descendante

- Assez simple à programmer sans faire appel à des automates.
  - Calcul de l'ensemble des premiers caractères dérivables à partir du membre droit d'une règle.
- Limitation des grammaires LL(1): grammaires peu naturelles.

Sommaire

Analyse ascendante

- L'analyse construit l'arbre de dérivation syntaxique en partant des feuilles.
- On garde en mémoire dans une pile une liste de non-terminaux et de terminaux correspondant à la portion d'arbre reconstruite.
- Deux opérations:
  - lecture/shift : on fait passer le terminal du mot à lire vers la pile
  - réduction/reduce : on reconnait sur la pile la partie droite d'une règle $X ::= x_1 \dots x_n$ et on remplace cette partie droite par X
- Le mot est reconnu si on termine avec le mot vide à lire et le symbole initial sur la pile.

Exemple de dérivation

	pile	entrée	action
1	ϵ	$id + id * id$	lecture
2	id	$+id * id$	reduction $E ::= id$
3	E	$+id * id$	lecture
4	$E+$	$id * id$	lecture
5	$E + id$	$*id$	reduction $E ::= id$
6	$E + E$	$*id$	lecture
7	$E + E*$	id	lecture
8	$E + E * id$	ϵ	reduction $E ::= id$
9	$E + E * E$	ϵ	reduction $E ::= E * E$
10	$E + E$	ϵ	reduction $E ::= E + E$
11	E	ϵ	reduction $S ::= E$
12	S	ϵ	succès

Décider de l'action

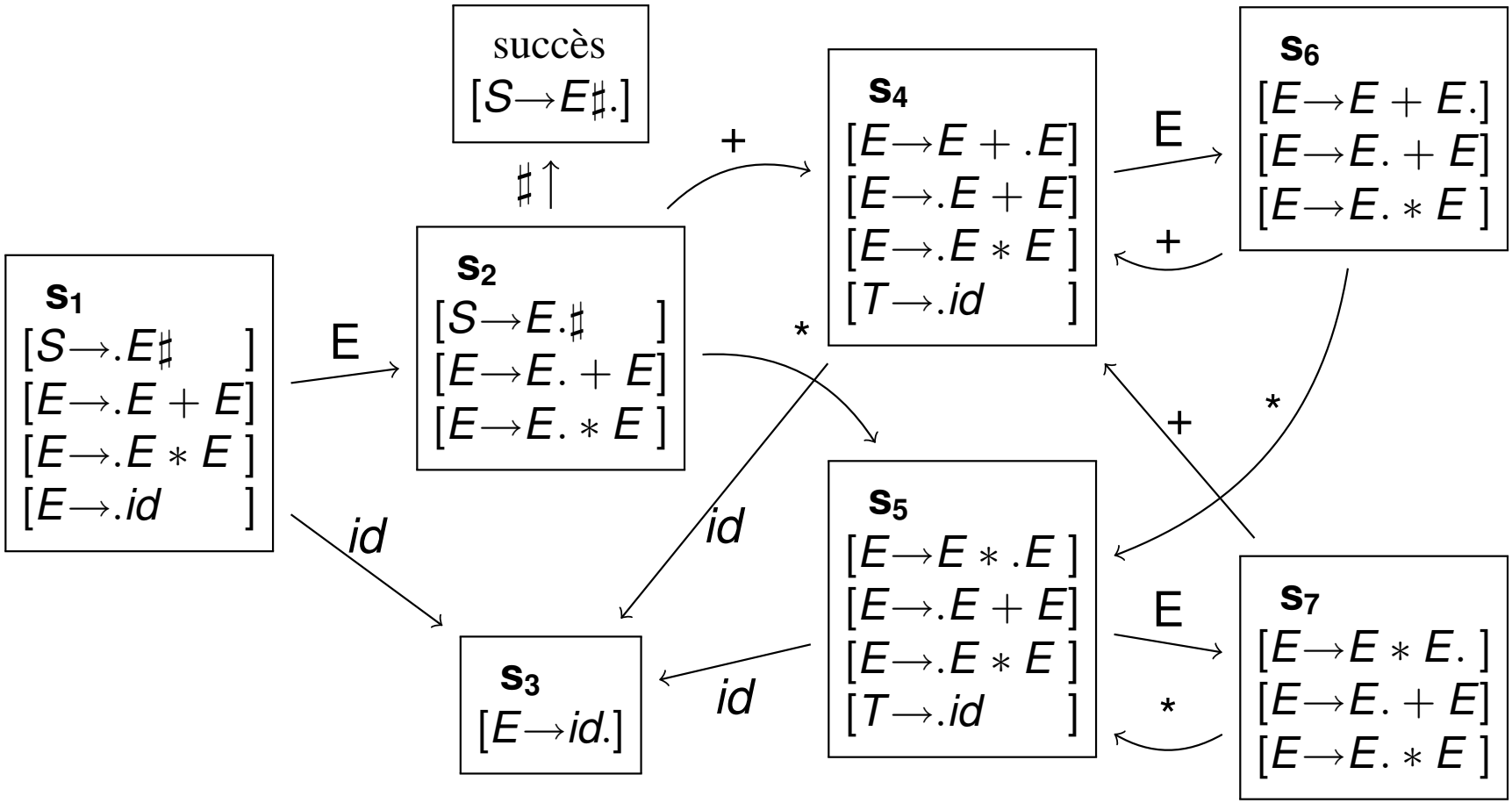
- A chaque état de la pile est associé un état d'un automate fini.
- Une table indique en fonction de l'état courant de la pile et du mot restant à analyser si on doit lire ou réduire.
  - Si lecture le nouvel état dépend de l'état courant et du caractère lu
  - Si réduction par $X ::= x_1 \dots x_n$ alors le nouvel état dépend de l'état avant la reconnaissance de $x_1 \dots x_n$ et de X.
- Chaque état correspond aux parties droites de règle qui pourraient être reconnues à ce point de l'analyse.

Exemple d'automate LR(0)

Grammaire :

S	::=	E ♯
E	::=	E + E
E	::=	E * E
E	::=	*id*

Automate :



Retour sur l'exemple

	pile	entrée	action
1	s_1	$id + id * id\sharp$	lecture
2	$s_1.id.s_3$	$+id * id\sharp$	reduction $E ::= id$
3	$s_1.E.s_2$	$+id * id\sharp$	lecture
4	$s_1.E.s_2.+.s_4$	$id * id\sharp$	lecture
5	$s_1.E.s_2.+.s_4.id.s_3$	$*id\sharp$	reduction $E ::= id$
6	$s_1.E.s_2.+.s_4.E.s_6$	$*id\sharp$	lecture
7	$s_1.E.s_2.+.s_4.E.s_6.*.s_5$	$id\sharp$	lecture
8	$s_1.E.s_2.+.s_4.E.s_6.*.s_5.id.s_3$	$\sharp$	reduction $E ::= id$
9	$s_1.E.s_2.+.s_4.E.s_6.*.s_5.E.s_7$	$\sharp$	reduction $E ::= E * E$
10	$s_1.E.s_2.+.s_4.E.s_5$	$\sharp$	reduction $E ::= E + E$
11	$s_1.E.s_2$	$\sharp$	reduction $S ::= E$
12	$s_1.S.\mathrm{succ}$	ϵ	succès

Actions possibles

- Si $s \xrightarrow{a} s'$ avec $a \in T$ alors on peut lire a à partir de s et passer en s'.
- Si s contient un item final $X \rightarrow x_1 \ldots x_n.$ alors on peut faire une réduction par la règle $X ::= x_1 \ldots x_n$.

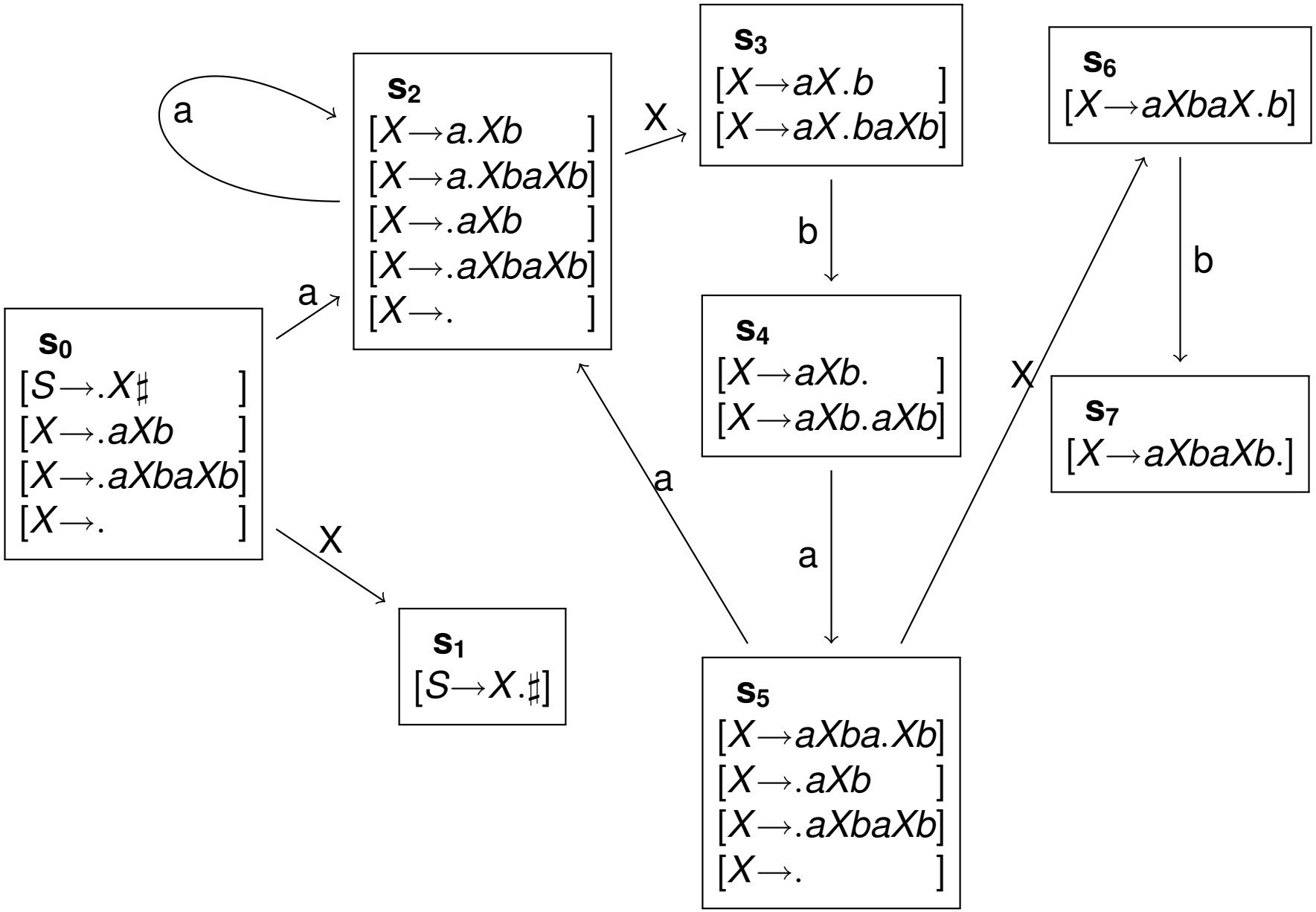
  On repart de l'état s' avant $x_1 \ldots x_n$ et on applique la transition $s' \xrightarrow{X} s''$.
- Conflits :
  - shift/reduce entre une lecture et une réduction
  - reduce/reduce entre deux réductions
- Les conflits peuvent parfois être résolus en identifiant les terminaux qui peuvent suivrent le non-terminal lié à la réduction
- Les conflits peuvent correspondre à des ambiguités dans la grammaire ou bien à des limitations de l'analyse.

Exemple

S	$::= X\ \sharp$
X	$::= aXb$
X	$::= aXbaXb$
X	$::= \epsilon$

On constate que les seuls terminaux qui suivent la dérivation d'un X sont $\{b, \sharp\}$.

Construction de l'automate



Sommaire

Précédences

- Les conflits reduce-reduce sont à éviter.
- Les conflits shift-reduce peuvent souvent être résolus à l'aide de précédences :
  - La règle a une précédence qui est par défaut celle du terminal le plus à droite mais qui peut être forcée (**%prec** *terminal*)
  - On compare la précédence de la règle *R* et du caractère à lire *c*:

`prec`(R) < `prec`(c)	lecture
`prec`(R) > `prec`(c)	réduction
`prec`(R) = `prec`(c)	associativité gauche : réduction
	associativité droite : lecture

Mise en œuvre ocamlyacc

- `ocamlyacc` indique le nombre te la nature des conflits
- `ocamlyacc -v` *parser.mly* permet d'avoir une trace de l'automate sous-jacent dans un fichier `parser.output`.

Exemple

Grammaire du langage ARITH sans indication de précédence.

```
 1  prog : instrs EOF

 2  instrs : instr
 3         | instrs instr

 4  instr : SET IDENT EQ expr
 5        | PRINT expr

 6  expr : CST
 7       | IDENT
 8       | expr PLUS expr
 9       | expr MINUS expr
10       | expr TIMES expr
11       | expr DIV expr
12       | MINUS expr
13       | LET IDENT EQ expr IN expr
14       | LP expr RP
```

Exemple d'état

```
state 14
        instr : PRINT expr .  (5)
        expr : expr . PLUS expr  (8)
        expr : expr . MINUS expr  (9)
        expr : expr . TIMES expr  (10)
        expr : expr . DIV expr  (11)

        PLUS  shift 21
        MINUS  shift 22
        TIMES  shift 23
        DIV  shift 24
        SET  reduce 5
        PRINT  reduce 5
        EOF  reduce 5
```

Exemple d'état avec conflit

```
29: shift/reduce conflict (shift 21, reduce 9) on PLUS
29: shift/reduce conflict (shift 22, reduce 9) on MINUS
29: shift/reduce conflict (shift 23, reduce 9) on TIMES
29: shift/reduce conflict (shift 24, reduce 9) on DIV
state 29
        expr : expr . PLUS expr  (8)
        expr : expr . MINUS expr  (9)
        expr : expr MINUS expr .  (9)
        expr : expr . TIMES expr  (10)
        expr : expr . DIV expr  (11)

        PLUS  shift 21
        MINUS  shift 22
        TIMES  shift 23
        DIV  shift 24
        SET  reduce 9
        IN  reduce 9
        PRINT  reduce 9
        EOF  reduce 9
        RP  reduce 9
```

Résolution des conflits en ocamlyacc

- reduce/reduce : la première règle est utilisée.
- shift/reduce :
  - utilisation des précédences ou de l'associativité;
  - si la règle et le token ont la même précédence et sont déclarés non associatifs alors c'est une erreur de syntaxe;
  - si pas de règle de précédence alors c'est l'action shift qui est choisie.

Il est conseillé de résoudre explicitement les conflits en modifiant la grammaire et/ou en indiquant les précédences

Syntaxe des fichiers ocamlyacc

```
%{
  header
  (* commentaire *)
%}
  declarations
  /* commentaire */
%%
  rules
  /* commentaire */
%%
  trailer
  (* commentaire *)
```

Exercice

I	::=	**if** E **then** I **else** I
I	::=	**if** E **then** I
I	::=	**skip**

- Montrer que cette grammaire est ambiguë.
- Quelle est la convention usuellement adoptée ?
- Modifier la grammaire pour la rendre non ambiguë.
- Comment indiquer des précédences sur la grammaire intiale pour lever l'ambiguïté ?

Solution

Exercice

$$
\begin{array}{lll}
\mathsf{E} & ::= & \mathsf{E}\ \mathsf{E} \\
\mathsf{E} & ::= & \mathsf{E} + \mathsf{E} \\
\mathsf{E} & ::= & \textbf{id}
\end{array}
$$

- Comment déclarer les précédences pour que l'application et l'addition associent à gauche et que l'application ait une précédence plus forte que l'addition.

Solution

Sommaire

Arbre de syntaxe abstraite

- Les constructions essentielles du langage
  - pas de construction pour les parenthèses
- Factoriser les constructions pour diminuer le nombre de cas à examiner
- reconnaitre et éliminer le “sucre syntaxique”
  ex: boucles for/while en C ou Java
- Localiser les erreurs

Localisation des erreurs

```
type location = Lexing.position * Lexing.position
type binop = Sum | Diff | Prod | Quot
type expr = { e_desc : loc_expr; e_loc : location}
type loc_expr =
    Cst of int
  | Var of string
  | Op of binop*expr*expr
  | Letin of string*expr*expr
type instr = { i_desc : loc_instr; i_loc : location}
type loc_instr = Set of string*expr | Print of expr
type prg = instr list
```

Vers l'analyse sémantique

L'arbre de syntaxe abstraite issu de l'analyse syntaxique correspond à l'entrée"brute" de l'utilisateur:
L'analyse sémantique va servir à glaner des informations:

- Relier les déclarations d'identifiant et leur utilisation.
  On utiliser alors des identifiants uniques ou bien des entiers pour plus d'efficacité.
- Calculer des informations de type sur chaque sous-expression.
- Distinguer certaines constructions (quand il y avait surcharge syntaxique) ou au contraire en éliminer (si redondance).
- ...

Remarque:

- Les différences entre les ast après analyse syntaxique et les ast après décoration par l'analyse sémantique peuvent justifier d'avoir deux types d'ast distincts (même s'ils partagent de nombreuses constructions).
- Si les deux structures sont suffisament similaires, on peut utiliser un même type ocaml éventuellement polymorphe.

Sommaire

Conclusion

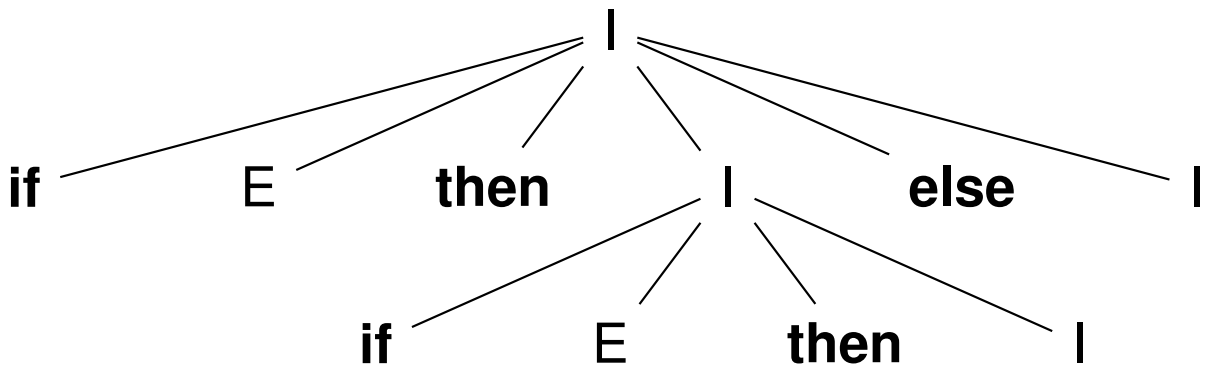
Déterminer ce qui peut/doit être réalisé à chaque étape:

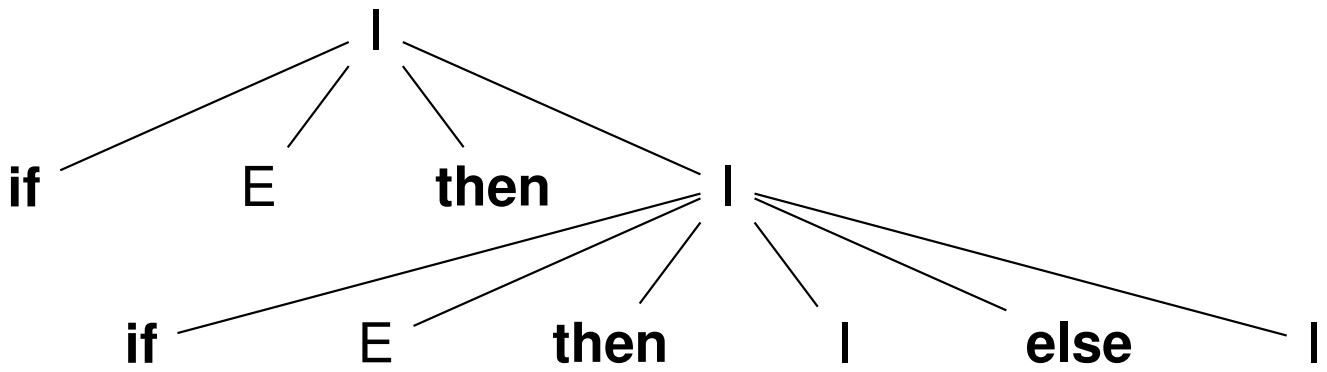
- Analyse lexicale : identifier ce qui joue un rôle dans la grammaire.
- Analyse syntaxique : la structure générale du programme.
- Analyse sémantique : les analyses *fines*.

Sommaire

Construction if-then-else

Ambiguité sur la phrase: **if** *E* **then if** *E* **then** *I* **else** *I*





Résolution du conflit

Grammaire non ambiguë

- expressions *IT* qui ont un **then** qui n'est pas associé à un **else**
- expressions *ITE* dont les **then** sont associés à des **else** qui peuvent aller entre **the** et un **else**.

$$
\begin{array}{lll}
I & ::= & IT \\
I & ::= & ITE \\
IT & ::= & \textbf{if } E \textbf{ then } I \\
IT & ::= & \textbf{if } E \textbf{ then } ITE \textbf{ else } IT \\
ITE & ::= & \textbf{if } E \textbf{ then } ITE \textbf{ else } ITE \\
ITE & ::= & \textbf{skip}
\end{array}
$$

Résolution du conflit

Précédences:

```
    10: shift/reduce conflict (shift 11, reduce 2) on ELSE
state 10
        instr : IF expr THEN instr . ELSE instr  (1)
        instr : IF expr THEN instr .  (2)

        ELSE  shift 11
        $end  reduce 2
```

Il faut choisir *shift* donc donner une précédence plus forte à **else** qu'à **then**.

Retour

précédence de l'application

```
%left PLUS
%nonassoc IDENT
%nonassoc app
%start expr
%type <unit> expr

%%

expr:
| expr expr %prec app { }
| expr PLUS expr      { }
| IDENT               { }
;
```

On regarde les différents cas de conflit possible

```
expr expr. PLUS       - reduce
expr expr. IDENT      - reduce
expr PLUS expr. PLUS  - reduce
expr PLUS expr. IDENT - shift
```