



UNIVERSITÉ DE BOURGOGNE  
ALGORITHME ET COMPLEXITÉ

par

JONATHAN ACEITUNO  
JOHAN JEGARD  
JULIEN VIREY

# CONSTRUCTION D'UN CONSTRUCTEUR D'ANALYSEUR SYNTAXIQUE ASCENDANT EN LANGAGE FONCTIONNEL

$\xrightarrow{A}$   $\Rightarrow$  poof!

Rapport rendu le 21 janvier 2010

Enseignants :

M. JEAN-JACQUES CHABRIER  
M. DOMINIQUE MICHELUCCI

# SOMMAIRE

SOMMAIRE	ii
LISTE DES FIGURES	iii
INTRODUCTION	1
1 MISE EN ABYME	2
1 HISTORIQUE DES COMPILATEURS	3
2 INTRODUCTION À L'ANALYSE SYNTAXIQUE	4
1 Vue d'ensemble	4
2 Grammaires reconnues	5
2 CONSTRUCTION D'UN PROGRAMME D'ANALYSE SYNTAXIQUE	8
1 GÉNÉRALITÉS SUR LE PROJET	9
1 Vue globale du programme	9
2 Utilisation du programme	9
3 Style fonctionnel et implantation d'algorithmes	10
2 UTILITAIRES ET REPRÉSENTATIONS OCAML	10
1 Types	10
2 Listes	12
3 Grammaire	13
3 GESTION DES DONNÉES	15
1 Exécution et arguments	15
2 Lecture d'un fichier de grammaire	16
3 Lecture sur l'entrée standard	17
4 ALGORITHME D'ANALYSE LR	17
1 Présentation générale	17
2 Implémentation avec Objective Caml	18
3 Construction de l'arbre syntaxique	18
5 CONSTRUCTION DES TABLES D'ANALYSE	20
1 Ensembles <i>Premier</i> et <i>Suivant</i>	20
2 Généralités sur l'analyse LR	23
3 Construction des tables d'analyse	28
CONCLUSION	33
A ANNEXES	35
1 PRÉ-REQUIS	36

2	UTILISATION . . . . .	37
3	EXEMPLES D'UTILISATION . . . . .	38
4	CAS AMBIGUËS . . . . .	39
	BIBLIOGRAPHIE . . . . .	42

## LISTE DES FIGURES

1	Un compilateur. . . . .	1
1.1	Grace Hopper en 1984 . . . . .	3
1.2	Une grammaire non LL(1) mais SLR . . . . .	5
1.3	Modèle d'un analyseur LR . . . . .	6
1.4	Les tables d'analyse SLR pour la grammaire de la figure 1.2 . . . . .	6
2.1	Algorithme de l'analyse LR <sup>[1]</sup> . . . . .	18
2.2	Algorithme de l'analyse LR pour Objective Caml . . . . .	19
2.3	Algorithme de remplissage jusqu'à ce que mort s'ensuive . . . . .	26
2.4	Algorithme de construction de la collection canonique des ensembles d'items . . . . .	28
A.1	Appel du programme depuis un terminal. . . . .	37
A.2	Arbre syntaxique pour l'expression " <i>nombre + nombre × nombre</i> " . . . . .	39
A.3	Arbre syntaxique pour l'expression " <i>(nombre + nombre) × nombre</i> " . . . . .	40
A.4	Arbre syntaxique pour l'expression <i>a a a a a</i> . . . . .	41
A.5	Conflit décaler/réduire pour la grammaire LR <i>if_then_else</i> . . . . .	41

# INTRODUCTION

UN compilateur est un programme qui lit un programme écrit dans un langage quelconque, appelé *langage source*, et le traduit en un programme similaire écrit dans un autre langage, le *langage cible*. Au cours de cette traduction, le compilateur remplit son rôle le plus important : signaler à l'utilisateur la présence d'erreurs dans le programme source.

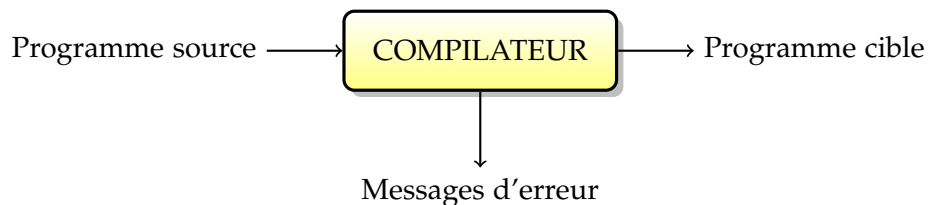


FIGURE 1 – *Un compilateur.*

Il existe un nombre impressionnant de langages sources, allant des langages de programmation traditionnels comme C ou Java aux langages spécialisés qui sont apparus dans quasiment tous les domaines d'application de l'informatique. Les langages cibles sont aussi variés : un langage cible peut-être un autre langage de programmation, ou bien le langage machine de n'importe quelle architecture.

La compilation se compose de deux parties essentielles : l'analyse et la synthèse. Le compilateur ne remplace donc pas les constructions du langage source directement par une construction du langage cible équivalente, il commence par analyser le langage source pour en construire une représentation intermédiaire, qu'il traduit ensuite en langage cible.

Pour ce projet, nous allons développer un programme construisant des analyseurs syntaxiques avec le langage *Objective Caml*.

# MISE EN ABYME



## SOMMAIRE

1	HISTORIQUE DES COMPILATEURS . . . . .	3
2	INTRODUCTION À L'ANALYSE SYNTAXIQUE . . . . .	4
1	Vue d'ensemble . . . . .	4
2	Grammaires reconnues . . . . .	5

**C**E premier chapitre présente un court historique de la compilation. Nous expliquerons ensuite les bases de l'analyse syntaxique.

## I Historique des compilateurs

Un programme est la manifestation d'une pensée qui s'exprime sous la forme d'un texte. Ce texte désigne des opérations qui doivent être réalisées par une machine qui exécute ainsi le texte. Comme le dit J.F. Perrot dans son cours de compilation de Paris 6 : *"un ordinateur est une machine à convertir la pensée en actions"*.

Un programme s'exprime dans un langage. Si ce langage est compris de la machine, cela signifie qu'elle peut exécuter directement le texte qui lui est proposé. Et au début de l'informatique, il fallait en effet programmer directement les ordinateurs en langage machine. La diversité des architectures n'arrangeait rien, car le code n'était pas du tout portable. La complexité de ce mode de programmation a très vite poussé les pionniers à utiliser les possibilités de l'informatique pour faciliter la programmation. Dans le cas d'un langage non compris par la machine, il est nécessaire de traduire le texte initial, le texte source, en un texte acceptable par la machine, le texte cible. L'opération conduisant à cette traduction s'appelle compilation.

Des langages de plus haut niveau ont alors commencé à voir le jour. En effet, les avantages d'utiliser les logiciels sur différentes architectures commençaient à être nettement plus élevés que le coût de développement d'un compilateur. Les premiers travaux sur la compilation se déroulèrent au début des années 50, ils étaient consacrés à la traduction de formules arithmétiques en code machine<sup>[1]</sup>. Le premier compilateur, A-0 System, fut écrit par Grace Hopper en 1951, pour le système Univac. Grace Hopper était une informaticienne amiral de la marine américaine. Elle est également la conceptrice du langage COBOL en 1959.



FIGURE 1.1 – Grace Hopper en 1984

C'est l'équipe FORTRAN, dirigé par John Backus, qui est souvent créditée pour avoir programmé le premier compilateur complet en 1957<sup>[6]</sup>. Pendant toutes les années 50, les compilateurs étaient considérés comme très complexes à écrire. La réalisation du compilateur FORTRAN prendra par exemple 18 années-homme de travail<sup>[2]</sup>.

Les connaissances sur la façon d'organiser et d'écrire des compilateurs se sont largement étendues depuis les années 50. On a découvert depuis des techniques systématiques pour traiter la plupart des tâches importantes qui sont effectuées pendant la compilation. Il existe

également des langages d'implémentation, des environnements de programmation et des outils logiciels bien adaptés au développement de compilateurs<sup>[2]</sup>.

## 2 Introduction à l'analyse syntaxique

Tout langage de programmation à des règles qui prescrivent la structure syntaxique des programmes bien formés. Pour obtenir des phrases syntaxiquement correctes d'un langage donné, il faut définir des règles d'assemblage syntaxique au moyen de grammaires. C'est un formalisme originaire de la linguistique que les mathématiciens théoriciens des langages et les informaticiens ont su reprendre avec profit.

### 1 Vue d'ensemble

Notre travail s'inscrit dans l'une des premières étapes de ce processus long et segmenté nommé compilation. La première étape, l'analyse lexicale, consiste à transformer une suite de caractères source en une suite de lexèmes (ou symboles terminaux). Elle est une étape à part entière car les lexèmes peuvent être définis de manière formelle par des expressions régulières, ainsi l'analyseur lexical reconnaît des langages de type 3, ou langages rationnels, à l'aide d'automates finis. La deuxième étape est l'analyse syntaxique. Elle consiste à transformer, grâce à un automate à pile, ce flot de lexèmes mal rasés en un arbre syntaxique parfumé, qui représente la syntaxe de la phrase d'entrée, c'est-à-dire son organisation par rapport à une grammaire non-contextuelle. C'est là que notre programme agit : partant d'un fichier dans lequel est décrite la grammaire sous forme d'un enchaînement de règles, et d'un flot d'entrée représentant une suite de lexèmes, le programme réalise la synthèse du fichier en une représentation abstraite de la grammaire puis effectue l'analyse syntaxique qui permet de construire peu à peu un arbre syntaxique qui sera retourné sous une forme ou une autre si la phrase est reconnue.

L'analyse syntaxique consiste à déterminer la chaîne de dérivations de l'axiome de la grammaire jusqu'à la phrase en entrée. Il y a deux façons d'aborder ce problème, et en conséquence on distingue deux écoles pour l'analyse syntaxique.

L'analyse descendante consiste à déterminer la chaîne des dérivations de la gauche vers la droite, de dérivations en partant de l'axiome et en développant les règles de la grammaire qui pourraient correspondre, jusqu'à arriver à la chaîne d'entrée. Un représentant de cette école est l'analyseur LL. Un autre est l'algorithme d'Earley<sup>1</sup>.

L'analyse ascendante consiste plutôt à partir de la chaîne d'entrée et d'essayer de reconstituer la chaîne de dérivations inverses pour arriver à l'axiome de la grammaire en réduisant à partir des règles de la grammaire. Un représentant de cette école est l'analyseur LR. Un autre est l'algorithme CYK.

Le monde de l'analyse syntaxique est tiraillé entre la performance (les analyseurs  $LL(k)$ )

<sup>1</sup>. On doit cet algorithme à Jay Earley qui l'a introduit dans sa thèse de doctorat avant de s'orienter vers la psychologie et le charlatanisme.

$$\begin{aligned} E &\longrightarrow Ea \\ E &\longrightarrow a \end{aligned}$$

FIGURE 1.2 – Une grammaire non LL(1) mais SLR

ou LR( $k$ )) et l'expressivité, ou l'étendue des grammaires reconnues (CYK ou Earley<sup>2</sup>), et chacune de ces caractéristiques doit être développée au détriment de l'autre.

## 2 Grammaires reconnues

### a Analyse LL(1)

L'analyse LL(1) ne reconnaît que les grammaires LL(1), et on peut ajouter que l'ensemble des langages LL(1) est exactement l'ensemble des langages reconnus par un analyseur LL(1). Il existe quelques grammaires qui posent problème à ce genre d'analyseur, et particulièrement les grammaires récursives à gauche. La figure 1.2 présente un exemple de grammaire non LL(1). L'analyse descendante présente une table d'analyse faisant correspondre un symbole d'entrée et un symbole non-terminal à une règle de la grammaire. Lorsque cette table comporte des entrées définies de façon multiple, alors la grammaire est ambiguë<sup>3</sup>. On exclut donc les grammaires ambiguës et les grammaires récursives à gauche de la famille des grammaires LL(1).

Néanmoins, il est possible de rendre LL(1) des grammaires qui ne le sont pas, en éliminant les récursivités à gauche et en factorisant, mais ce procédé rend plus difficile la création de la grammaire que celle de l'analyseur correspondant, et la grammaire résultante est plus difficile à lire.

### b Analyse ascendante

L'univers des analyseurs ascendants comporte, entre autres, la famille des LR. Ce qui diffère est que les dérivations sont construites à partir de la droite, en lisant toujours de gauche à droite. En réalité l'analyseur va procéder par des réductions et par décalages du tampon d'entrée. Ces deux actions principales sont à l'origine du nom "analyseur par décalage/réduction" communément donné aux analyseurs ascendants. Le principe de fonctionnement de l'automate est le même pour tous les membres de la famille LR, et ce sont les tables d'analyse qui changent. La figure 1.3 présente le fonctionnement d'un analyseur LR. Un exemple de table d'analyse pour la grammaire de la figure 1.2 est disponible à la figure 1.4. Le symbole \$ est celui qui indique la fin de la chaîne.

2. Malgré des conditions nécessaires pour ces algorithmes, par exemple la grammaire qui doit être mise sous forme normale de Chomsky pour CYK.

3. Cette définition est à étendre à tout analyseur.

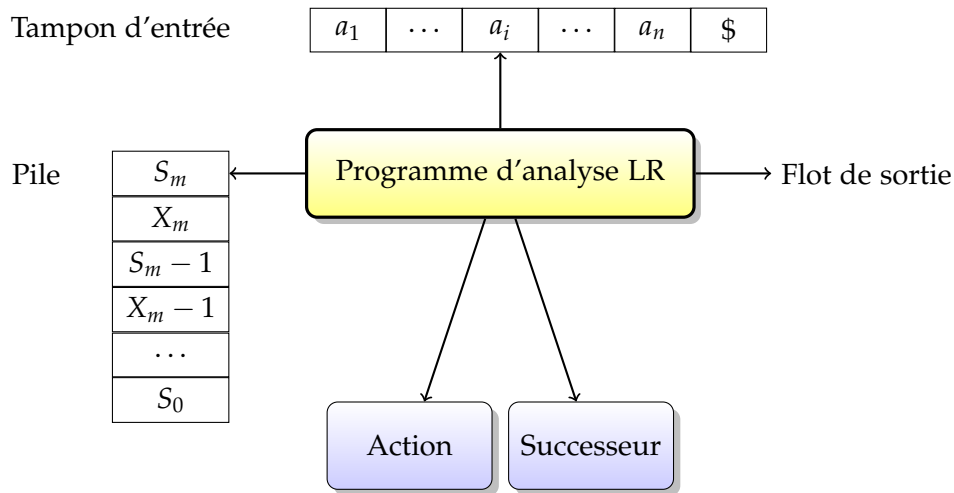


FIGURE 1.3 – Modèle d'un analyseur LR

État	Action		Successeur
	a	\$	
0	Décaler et passer à 2		1
1	Décaler et passer à 3	Accepter	
2	Réduire par $E \rightarrow a$	Réduire par $E \rightarrow a$	
3	Réduire par $E \rightarrow Ea$	Réduire par $E \rightarrow Ea$	

FIGURE 1.4 – Les tables d'analyse SLR pour la grammaire de la figure 1.2

Ces tables sont au nombre de deux et sont nommées *Action* et *Successeur* :

- La table *Action* indique une action à accomplir entre “décaler et passer à l'état  $n$  de l'automate”, “réduire par la production  $P$  de la grammaire”, “accepter la phrase en entrée” et “produire une erreur”, étant donné l'état actuel et le symbole à lire en entrée.
- La table *Successeur* indique simplement à quel état aller lorsqu'un non-terminal donné a été reconnu.

L'analyseur qui produit les tables les plus simples pour une grammaire est l'analyseur SLR. Néanmoins, on peut dire qu'il n'est pas très intelligent, car même si une grammaire n'est pas ambiguë elle peut ne pas être SLR, du fait que l'analyseur ne mémorise pas assez d'éléments de contexte pour pouvoir décider d'une action ou d'une autre.

Un analyseur suffisamment puissant pour pouvoir exprimer des langages de programmation est l'analyseur LR(1). C'est un analyseur bien plus “intelligent” que l'analyseur SLR parce qu'il génère plus d'états.

Frank DeRemer, pensant qu'il y a même trop d'états, a eu l'idée géniale d'inventer l'analyseur LALR<sup>4</sup>. Une version moderne de cet analyseur reprend les tables d'analyse LR(1) en factorisant tout ce qui est factorisable, c'est-à-dire ici en regroupant les états similaires. Ainsi les tables LALR et les tables SLR ont le même nombre d'états. Il y a relativement peu

4. En réalité, c'est lui aussi qui a inventé l'analyseur SLR, mais le fait qu'il aime convertir les gens au créationnisme nous invite à parler de lui avec une impartialité toute relative.

de grammaires LR(1) qui ne sont pas reconnues par les analyseurs LALR, et c'est le type d'analyseur que construit le programme yacc.

En ces temps de discorde, les idées du passé vacillent, et si le Dragon recommande de ne pas utiliser  $k > 1$  dans les analyses LL( $k$ ) et LR( $k$ ) car ce n'est pas jugé utile, d'autres personnes<sup>[7]</sup> ne sont pas de cet avis, grognent contre les gens qui assimilent un traducteur à un simple analyseur syntaxique et prouvent que l'existence d'analyseurs avec  $k$  symboles de pré-vision simplifierait la conception des traducteurs.

# CONSTRUCTION D'UN PROGRAMME D'ANALYSE SYNTAXIQUE

# 2

## SOMMAIRE

1	GÉNÉRALITÉS SUR LE PROJET	9
1	1 Vue globale du programme	9
2	2 Utilisation du programme	9
3	3 Style fonctionnel et implantation d'algorithmes	10
2	UTILITAIRES ET REPRÉSENTATIONS OCAML	10
1	1 Types	10
2	2 Listes	12
3	3 Grammaire	13
3	GESTION DES DONNÉES	15
1	1 Exécution et arguments	15
2	2 Lecture d'un fichier de grammaire	16
3	3 Lecture sur l'entrée standard	17
4	ALGORITHME D'ANALYSE LR	17
1	1 Présentation générale	17
2	2 Implémentation avec Objective Caml	18
3	3 Construction de l'arbre syntaxique	18
5	CONSTRUCTION DES TABLES D'ANALYSE	20
1	1 Ensembles <i>Premier et Suivant</i>	20
2	2 Généralités sur l'analyse LR	23
3	3 Construction des tables d'analyse	28

**C**E chapitre détaille les étapes de la construction d'un programme d'analyse syntaxique ascendant de type LR. Voici donc *DAHU* (Dahu's an Analyser Handling Unix).

## I Généralités sur le projet

Nous avons écrit notre programme avec Objective Caml : ce langage permet de programmer en mélangeant différents paradigmes, dont la programmation impérative, fonctionnelle et par objets. Nous avons préféré adopter un style plutôt fonctionnel, en utilisant le plus possible les types de base et en décomposant le programme par fonctions. Malheureusement, nous n'avons pas pu nous offrir le luxe d'avoir réalisé le projet dans un style fonctionnel pur<sup>1</sup>.

### 1 Vue globale du programme

Le programme se nomme DAHU<sup>2</sup>. Il faut lui fournir un fichier contenant une grammaire et une expression à analyser pour qu'il puisse travailler correctement. Une brève analyse découpe ensuite ce fichier de grammaire afin d'en faire une représentation interne abstraite dans le but de construire les tables d'analyse syntaxique. Le programme peut enfin procéder à l'analyse syntaxique selon le type d'analyse choisi. Une suite d'instruction pour le programme *DOT* est finalement balancée sur la sortie standard, prête à subir toutes les redirections les plus folles, comme celle de *DOT*.

### 2 Utilisation du programme

Cette section ne prétend pas remplacer le manuel du programme mais apporte quelques informations utiles sur l'utilisation de notre programme.

#### a Fichier de grammaire

Le fichier de grammaire est composé :

- d'une première ligne comportant le texte `slr`, `lr` ou `lalr` ;
- de plusieurs autres lignes dont chacune représente une production pour la grammaire décrite.

Chaque production de la grammaire est composée d'une suite de caractères sans espace représentant le symbole de la partie gauche, puis de deux points, et enfin d'une suite de chaînes de caractères séparées par des espaces représentant les symboles de la partie droite de la production.

Par convention, le premier symbole rencontré lors de la lecture du fichier de grammaire est l'axiome de la grammaire.

#### b Insertion du programme dans un flot d'exécution

Nous avons fait en sorte que notre programme fasse le moins de choses possibles afin de laisser toute latitude à l'utilisateur pour choisir le devenir de ce qu'il produit. Ainsi, notre programme lit une phrase à analyser sur l'entrée standard ou sur les arguments restants, et sort un code DOT représentant un arbre syntaxique sur la sortie standard. Lorsqu'une erreur survient, elle est affichée sur la sortie d'erreur standard et le programme se termine avec un code de retour anormal. Ainsi il est extrêmement facile d'utiliser ce programme pour atteindre

1. Et ce malgré quelques recherches et maux de tête sur la question des monades !

2. DAHU est un acronyme récursif qui signifie : DAHU, un analyseur syntaxique qui sent le yacc !

divers buts. Par exemple, la lecture du fichier `codesource.s` et l’affichage de l’arbre syntaxique dans une fenêtre GTK+ seront effectués en une ligne de shell script :

```
1 cat codesource.s | dahu fichier_grammaire | dot -vTgtk
```

### 3 Style fonctionnel et implantation d’algorithmes

Nous avons quelque peu perdu nos repères au début du projet car nous devions mettre en œuvre des algorithmes exprimés de manière à être implantés dans un style procédural, abordés avec plus ou moins de détail concernant les structures de données et le style à employer<sup>[1]</sup>. Au lieu de cela, nous avons utilisé des fonctions récursives à tour de bras mais il est clair que si la programmation fonctionnelle n’est pas la tasse de thé de tout un chacun, elle a l’avantage de permettre de produire un code plus compact et bien plus proche des spécifications de l’algorithmique, puisqu’on est moins concentré sur les problèmes de gestion de la mémoire et que presque tout le code écrit concerne directement l’algorithme implémenté.

## 2 Utilitaires et représentations Ocaml

Cette section résume les fonctions et les types utilitaires que nous avons dû écrire. Les types nous ont permis de modéliser les données à traiter de façon claire. Ces fonctions sont utiles uniquement pour traiter les structures de données mais ne concernent pas directement le sujet. Nous ne décrirons pas les algorithmes lorsqu’ils sont triviaux ou lorsque nous jugeons le code assez explicite. Nous utilisons le style fonctionnel pour écrire ces fonctions.

### 1 Types

#### a Symbole

Nous avons tout d’abord voulu représenter le type symbole, qui est celui de tous les éléments qui peuvent être contenus dans une grammaire. Ce type peut être soit  $\epsilon$ <sup>3</sup>, soit égal au symbole \$, soit un autre symbole dont on précisera alors la chaîne de caractères représentative.

```
1 type symbole = Epsilon | Dollar | S of string
2 # S" id";;
3 - : symbole = S " id"
```

#### b Production

Ensuite, il a fallu définir le type production. Une production est une règle de la grammaire, elle est composée d’un non-terminal à gauche, suivi d’une liste de symboles à droite. Sachant que le type symbole a été défini précédemment, la représentation OCaml de ce type peut s’écrire ainsi :

3. Devant les multiples sources qui considèrent  $\epsilon$  une fois comme la chaîne vide, et une autre fois comme un caractère spécial, et surtout en considérant que Objective Caml est très pointilleux lorsqu’il s’agit de respecter les types des données, nous avons choisi de considérer que  $\epsilon$  était un symbole, et que  $\epsilon$  en tant que chaîne vide serait représenté par [].

```
1 type production = symbole * (symbole list);;
```

La règle suivante :

$$E \rightarrow E + T$$

donnera ceci :

```
1 S"E", [S"E";S"+";S"T"]
```

### c Grammaire

La suite coule de source, car le type grammaire est une liste de productions :

```
1 type grammaire = production list;;
```

### d Tables d'analyse

Nous avons également créé un type qui modélise les tables d'analyse, indispensables à l'analyse syntaxique, un type action pour répertorier les actions possibles, ainsi qu'un type tableAction et tableSuccesseur dont les commentaires parlent d'eux-mêmes.

```
1 (* Les tables d'analyses identifient un élément par un état et un
   2   symbole *)
   2 type coupleTable = int * symbole;;
   3
   4 (* Toutes les actions possibles *)
   5 type action = Erreur | Accepter | DecalerEtEmpilerEtat of int |
   6   ReduireParProduction of int;;
   7
   8 (* Une table d'action associe à un coupleTable une action *)
   8 type tableAction = coupleTable * action;;
   9
  10 (* Une table de successeur associe à un coupleTable un successeur
     *)
  11 type tableSuccesseur = coupleTable * int;;
```

### e Arbre

Puis, nous avons finalement créé un type arbre qui va s'avérer d'une grande utilité pour la construction de l'arbre syntaxique.

```
1 type arbre= Sommet of symbole * arbre list;;
```

## 2 Listes

Nous avons eu besoin d'écrire quelques opérations afin de rendre l'utilisation de listes plus agréable.

### a Ajout d'un élément à une position donnée

Il s'agit d'une fonction récursive qui parcourt la liste et la reconstruit en attendant patiemment l'heure d'insérer le fatidique élément, en fonction d'une position.

```

1 let rec ajouter_a i l x = if i<0 then l else match l with
2   | [] -> [x]
3   | t::q -> if i=0 then [x;t] @ q
4             else t::(ajouter_a (i-1) q x);;

```

### b Transformation d'une liste en ensemble

Nous reconstruisons ici la liste par récursion gauche grâce à la fonction `fold_left`, en ajoutant ou non un élément en fonction de son appartenance au reste de la liste.

```

1 let rec enlever_doublons liste = List.fold_left (
2   fun l x -> if List.mem x l then l else l @ [x]
3 ) [] liste;;

```

### c Différence de deux ensembles

L'ambiguïté de notre représentation fait qu'il suffit, pour faire  $A - B$ , d'enlever les éléments qui peuvent appartenir à la liste B dans la liste A.

```

1 let rec subtract l1 l2 = match l1 with
2   | [] -> []
3   | t::q -> if List.mem t l2 then subtract q l2
4             else t::(subtract q l2);;

```

### d Position d'un élément dans une liste

Nous avons eu besoin d'obtenir la position d'un élément donné dans une liste. Il s'agit, en parcourant cette liste, de repérer une égalité structurelle entre l'élément courant et l'élément donné.

```

1 let index_of e l =
2   let rec index_of_r e l n = match l with
3     | [] -> -1
4     | t::q when t = e -> n
5     | t::q -> index_of_r e q (n+1)
6   in index_of_r e l 0;;

```

**e Positions d'un élément dans une liste**

Nous avons également eu besoin d'obtenir toutes les positions d'un élément donné dans une liste, au cas où l'élément apparaît plusieurs fois. Notons que nous devons utiliser une liste qui représente bel et bien une liste pour que cette opération ait un sens différent de la précédente.

```

1 let indices_of e l =
2   let rec indices_of_r e l n = match l with
3     | [] -> []
4     | t::q when t = e -> n::(indices_of_r e q (n+1))
5     | t::q -> indices_of_r e q (n+1)
6   in indices_of_r e l 0;;

```

**f Sous-liste d'une liste**

Nous avons finalement eu envie d'extraire une sous-liste d'une liste. Cette fonction extrait la sous-liste de l'indice  $i$  à l'indice  $j$  exclu.

```

1 let rec sublist l deb fin = match l with
2   | [] -> []
3   | t::q when deb > 0 -> sublist q (deb-1) (fin-1)
4   | t::q when fin > 0 -> t::(sublist q 0 (fin-1))
5   | t::q -> [];;

```

**3 Grammaire**

Nous avons eu besoin de quelques fonctions bien pratiques pour extraire des informations d'une grammaire.

**a Trouver l'axiome d'une grammaire**

L'axiome d'une grammaire est la partie gauche de la première production.

```

1 let axiome g = fst (List.hd g);;

```

**b Obtenir toutes les productions dont la partie gauche est un non-terminal donné**

Il suffit de parcourir récursivement la grammaire et de sortir les productions dont la partie gauche est le non-terminal voulu.

```

1 let rec que_faire_avec g r = match g with
2   | [] -> []
3   | t::q -> if fst t = r then
4     t::(que_faire_avec q r) else que_faire_avec q r;;

```

**c Obtenir toutes les productions dont la partie droite contient un symbole donné**

Ce qui rend la lecture de cette fonction difficile est le fait qu'elle sort une liste de quadruplets composés d'une production, de la position du symbole donné dans cette production, de la chaîne avant ce symbole dans cette production et de la chaîne après ce symbole dans cette production.

```

1 let rec ou_est s g = match g with
2   | [] -> []
3   | (nt, ls)::q -> let idx = indices_of s ls in (List.map (
4     function i -> (
5       (nt,ls), i, sublist ls 0 i,
6       sublist ls (i+1) (List.length ls))
7     ) idx) @ (ou_est s q);;
```

**d Obtenir l'ensemble des symboles dans une grammaire**

Aussi trivial que peu performant :

```

1 let symboles gr = let rec tous_les_symboles g = match g with
2   | [] -> []
3   | t::q -> match t with (nnterm, l) ->
4     nnterm::l @ (tous_les_symboles q)
5   in enlever_doublons (tous_les_symboles gr);;
```

**e Obtenir l'ensemble des non-terminaux dans une grammaire**

On reconnaît un non-terminal au fait qu'il apparaît au moins dans la partie gauche d'une production.

```

1 let rec nonterminaux g = enlever_doublons (List.map fst g);;
```

**f Obtenir l'ensemble des terminaux dans une grammaire**

Lorsqu'on sait obtenir l'ensemble des symboles ainsi que les non-terminaux, étant donné que l'ensemble des terminaux et celui des non-terminaux forment une partition de l'ensemble des symboles, la suite n'est pas plus difficile.

```

1 let terminaux g = subtract (symboles g) (nonterminaux g);;
```

**g Augmenter une grammaire**

Cette opération ne se révèle utile que lors de la construction des tables d'analyse. Sachant récupérer l'axiome d'une grammaire, on peut l'augmenter. Augmenter une grammaire  $G$  où l'axiome est  $S$  revient à donner la grammaire augmentée  $G' = G \cup \{S' \rightarrow S\}$ .

```

1 let augmenter g = (S((string_of_symbole (axiome g)) ^ "'"),
2   [axiome g])::g;;
```

**h** Obtenir le non terminal à la position voulue

Retourne tout simplement le non-terminal de gauche se trouvant à la position voulue dans la grammaire. Utile dans l'algorithme d'analyse.

```
1 let nonterminalAt pos g= List.nth (List.map fst g) pos ;;
```

**i** Nombre de symboles dans la partie droite d'une règle

Petite fonction simple mais néanmoins utile lors de l'analyse.

```
1 let nbSymbolePartieDroite p= List.length (snd p) ;;
```

## 3 Gestion des données

Dans l'optique de réaliser une application en ligne de commande se comportant comme un bon citoyen du monde UNIX, nous avons réalisé des fonctions pour que notre application se comporte à la manière du programme `cat`. Notre application se devait alors d'être simple, tout en restant paramétrable. Une application en ligne de commande possède traditionnellement<sup>4</sup> un aide-mémoire, assorti d'une courte description, du détail des options disponibles et d'exemples d'utilisation. Le passage d'argument est donc géré, et plusieurs options permettent de paramétrer le type de retour. Le programme fonctionne aussi bien avec le passage d'arguments qu'avec l'entrée standard (on peut donc considérer cette possibilité comme un mode interactif).

### 1 Exécution et arguments

En exécutant le programme tel quel, le `dahu` renvoie une erreur précise à l'utilisateur : le programme a besoin qu'on lui précise un fichier de grammaire pour fonctionner.

```
1 usage: dahu [options] fichier_grammaire [token [token1] [...]]
2 Pour en savoir davantage, faites : « dahu -h ».
```

Pour en savoir davantage il faut donc utiliser l'option "-h", qui permet d'afficher l'aide mémoire.

```
1 _____
2 | _ _ \ _ | |
3 | | | | _ _ | | _ _
4 | | | / _ \ | ' _ \ | | |
5 | | / / ( _ | | | | | _ |
6 | ___ / \ _ , _ | | | \ _ , _ |
7
8 DAHU : Dahu's an Analyser Handling Unix
9
```

4. Tout du moins chez les systèmes d'exploitation respectables...

```
10 utilisation
11 -----
12     dahu [options] fichier_grammaire [token [token1] [...]]
13
14 Informations
15 -----
16     PWD : /home/test
17     Version Ocaml : 3.11.1
18     OS : Unix
19
20 Options
21 -----
22     -h      Affiche l'aide mémoire.
23     -a      Affiche les auteurs du programme.
24     -v      Affiche la version du programme.
25
26 Documentation
27 -----
28     Analyseur syntaxique écrit en Ocaml qui réalise une analyse LR.
29     Le programme utilise le fichier de grammaire entré en paramètre
30     pour sortir un arbre syntaxique en fonction de l'expression
31     spécifiée.
32
33 Exemple
34 -----
35     dahu grammaire nombre + nombre - nombre
```

Une utilisation classique a déjà été abordée. Pour rappel, on appelle donc le logiciel en précisant le nom du fichier de grammaire, suivi de l'expression à interpréter. Il faut par contre faire attention à l'emploi des symboles utilisés par le shell : `* ! ? { }` ... seront interprétés comme des expressions régulières par Unix. Pour utiliser ces symboles, il faut soit encadrer l'expression à interpréter par de simples quotes `'expression'`, soit utiliser l'entrée standard (par exemple, en faisant un *here-document*).

## 2 Lecture d'un fichier de grammaire

La grammaire à utiliser doit être stockée dans un fichier texte. Les lexèmes du fichier de grammaire doivent être séparés uniquement par des espaces, et la première ligne représente le type d'analyse à effectuer. Pour utiliser le fichier, il suffit de fournir son nom au programme, ou son chemin absolu si le fichier n'est pas dans le répertoire de travail. Le fichier est ensuite traduit par la fonction *lire*, qui analyse ligne par ligne le fichier. Chaque ligne est divisée en type *symbole*, et concaténée dans une liste dont la tête représente le symbole terminal. Cette liste est représentée dans notre programme par le type *production*, qui est un couple d'un *symbole* et d'une liste de *symbole*. Finalement, la grammaire est représentée par une liste de productions.

### 3 Lecture sur l'entrée standard

Si le programme est appelé avec seulement un fichier de grammaire en argument, l'expression peut-être saisie sur l'entrée standard. Le délimiteur est l'espace ou le retour chariot. Les expressions suivantes sont donc équivalentes :

```
1 nombre + nombre - nombre * nombre
```

```
1 nombre ↵
2 + ↵
3 nombre ↵
4 - ↵
5 nombre ↵
6 * ↵
7 nombre ↵
```

Pour quitter l'entrée standard, il faut utiliser le mot clef `QUIT` ou la combinaison de touches `Ctrl+D`. Chaque lexème de l'expression est concaténé dans une liste.

## 4 Algorithme d'analyse LR

### 1 Présentation générale

L'algorithme principal de l'analyse LR est relativement simple à comprendre. Il prend en entrée une chaîne correspondant à la phrase que l'on souhaite vérifier ainsi que les tables d'analyse pour une grammaire donnée (les tables d'actions et de successeurs dont la construction sera présentée dans la section suivante).

Le résultat retourné est soit une erreur, si la phrase n'est pas dans le langage engendré par la grammaire, soit une analyse ascendante de la phrase sous forme d'un arbre. Pour ce faire, on utilise une pile initialisée à 0 dans laquelle on ajoute les états correspondants aux indications fournies par les tables d'analyse : soit on décale et on empile un nouvel état, soit on réduit par une règle et on empile la valeur du champ de la table successeur correspondant au couple du non-terminal par lequel on a réduit et de l'état courant dans la pile.

On peut également décider d'empiler non seulement l'état, mais aussi le symbole correspondant soit au symbole lu (lors du décalage), soit au non-terminal par lequel on a réduit. La figure 2.1 détaille l'algorithme de l'analyse LR, dans le cas d'une pile à états seulement.

Quelques rappels :

- $Action(x, y)$  désigne l'action répertoriée dans la table d'analyse pour un état  $x$  et un symbole  $y$  (décaler, réduire ou accepter).
- $Successeur(x, y)$  désigne le successeur répertorié dans la table d'analyse pour un état  $x$  et un symbole non terminal correspondant à celui par lequel on va réduire le(s) symbole(s) courant(s).
- $V$  désigne un symbole non-terminal, tandis que  $s$  désigne un symbole terminal et  $\alpha$  ou  $w$  désignent une chaîne composée de terminaux et/ou de non-terminaux.

```

1  initialiser le pointeur p sur le premier symbole de la chaine à
   analyser
2  begin boucler indéfiniment
3      soit s l'état en sommet de pile et a le symbole pointé par p;
4      si Action[s,a]=décaler s' alors begin
5          empiler s'; (*s' est le nouvel état en sommet de pile*)
6          avancer p sur le prochain symbole en entrée;
7      end
8      sinon si Action[s,a]=réduire par V->w alors begin
9          soit s' le nouvel état en sommet de pile;
10         empiler Successeur[s',V];
11         imprimer éventuellement une identification de la production
           V->w
12     end
13     sinon si Action[s,a]=accepter alors
14         retourner; (*Fin du programme, la grammaire est reconnue *)
15     sinon Erreur();
16 end

```

FIGURE 2.1 – Algorithme de l'analyse LR<sup>[1]</sup>.

## 2 Implémentation avec Objective Caml

Objective Caml est un langage optimisé pour les implémentations fonctionnelles, c'est à dire pour un rythme rapide d'allocation/libération mémoire de petits objets, ce qui n'a donc pas d'impact sensible sur les performances des programmes. Il était donc intéressant de se pencher sur ce type d'implémentation, afin de changer de la routine procédurale<sup>5</sup>. Pour ce faire, il a d'abord fallu utiliser une série de fonctions utiles au bon fonctionnement, mais aussi à la lisibilité du programme (cf. section 3). Ensuite, comment transcrire une boucle infinie qui attend une valeur de retour en langage fonctionnel? En utilisant la récurrence, bien évidemment! Nous avons donc fait en sorte que la fonction se rappelle elle-même, avec les bons arguments, jusqu'à son aboutissement.

## 3 Construction de l'arbre syntaxique

C'est maintenant que la tâche se complique : il s'agit de pouvoir imprimer le graphe, à partir des résultats de l'algorithme, dans le langage DOT sur la sortie standard. Pour cela, nous avons créé deux fonctions qui s'exécutent à chaque nouvel appel à l'algorithme principal. Le principe est le suivant : on commence avec une liste d'arbres vide, et lorsque l'on passe par l'action décaler, on rajoute alors l'arbre composé du symbole lu en entré dans la liste d'arbres. Lorsque l'on passe par l'action réduire, on compte le nombre de symboles de la partie droite de la production par laquelle on va réduire, soit  $n$  ce nombre, on enlève alors  $n$  arbres de la liste d'arbres avant de combiner ces  $n$  arbres en un seul et nouvel arbre dont le sommet est le non-terminal de gauche de la règle par laquelle on a réduit.

---

5. Ceci est un pléonasme dissimulé.

```
1 let rec analyseLR liste pile tableAction tableSuccesseur grammaire
  arbrelist =
2   match liste with
3     | []->[]
4     | a::q-> match action (List.hd pile, a) tableAction with
5     | DecalerEtEmpilerEtat(n)-> analyseLR q (n::pile)
        tableAction tableSuccesseur grammaire (decaler_arbre
        arbrelist a)
6     | ReduireParProduction(n)->
7       let nbsymb= nbSymbolePartieDroite (List.nth grammaire (
8         n)) in
9       let nvPile= retirerT nbsymb pile in
10      let nvEtat=successeur (List.hd nvPile, (nonterminalAt (
11        n)grammaire))tableSuccesseur in
12        let nt=(nonterminalAt (n)
13          grammaire) in
          analyseLR liste (nvEtat::nvPile) tableAction
          tableSuccesseur grammaire (reduire_arbre arbrelist
          nt nbsymb)
11     | Accepter-> arbrelist
12     | Erreur->failwith "Erreur_de_syntaxe,_vérifiez_votre_
13       saisie"
      ;;
```

FIGURE 2.2 – Algorithme de l'analyse LR pour Objective Caml

Voici ces deux fonctions écrites en Objective Caml :

```

1 let decaler_arbre arbrelist symbole=
2     match arbrelist with
3     | []->[Sommet (symbole, [])]
4     | _->arbrelist@[Sommet (symbole, [])];;
5
6 let reduire_arbre arbrelist sommet nb=
7     match arbrelist with
8     | []->failwith "Erreur, _liste_vides!"
9     | _->let subl=sublist arbrelist (List.length (arbrelist) -nb
10        ) (List.length arbrelist) in
11        let newlist=retirerQ (nb) (arbrelist) in
            newlist@[Sommet (sommet, subl)];;

```

A la fin, il ne reste plus qu'un seul et unique arbre dans la liste, composé de tous les sommets utiles à la conception du graphe final. Il faut maintenant imprimer ce graphe dans la syntaxe DOT. Il a donc fallu créer une fonction imprimer qui va prendre en argument un arbre syntaxique et imprimer tous ces sommets reliés en langage DOT. Le problème qui s'est posé est celui des doublons : il a fallu déterminer lorsqu'un même symbole était unique ou lorsqu'on devait le dupliquer en utilisant les "labels" de DOT. On donne alors un nom unique à chaque symbole en faisant rentrer des entiers dans la fonction (par exemple le premier E est E0x0) car la syntaxe DOT demande évidemment des noms différents pour pouvoir tracer chaque liaison. Puis on détermine quelle étiquette donner à ce symbole afin que le nom du symbole apparaisse sous sa forme simple à l'utilisateur.

## 5 Construction des tables d'analyse

Nous avons choisi d'implanter plusieurs méthodes d'analyse syntaxique, pour que l'utilisateur du programme puisse éventuellement utiliser ce dernier à des fins de test dans son apprentissage de la compilation. Ainsi, notre programme comporte tout ce qu'il faut où il faut pour réaliser une analyse SLR, une analyse LR ou une analyse LALR.

### 1 Ensembles *Premier* et *Suivant*

La construction des tables d'analyse pour l'analyse LL se base sur l'existence de deux fonctions *Premier* et *Suivant*. Notre programme n'est pas équipé pour proposer la construction d'un analyseur LL(1), mais il suffirait de peu car les deux fonctions sus-citées sont la clé de son fonctionnement. Nous utilisons ces fonctions, et plus particulièrement la fonction *Suivant* lors de l'analyse SLR. Pour une chaîne  $\alpha$ ,  $Premier(\alpha)$  est l'ensemble des terminaux pouvant débiter une dérivation de  $\alpha$ . Pour un non-terminal  $A$ ,  $Suivant(A)$  définit l'ensemble des terminaux qui peuvent apparaître immédiatement à droite de  $A$  dans une dérivation.

#### a Construction des ensembles *Premier*

Nous avons reformulé la définition par rapport à ce que nous connaissons<sup>[1] [3]</sup> afin qu'un algorithme récursif de calcul puisse être automatiquement déduit.

Afin de décrire le fonctionnement de  $Premier(\alpha)$ , on distinguera plusieurs cas selon la nature de  $\alpha$  :

- quand  $\alpha \in \{\emptyset, \{\epsilon\}\}$  alors :

$$Premier(\alpha) = \{\epsilon\}$$

- quand  $\alpha = \{a\}$ ,  $a$  terminal, alors :

$$Premier(\alpha) = \{a\}$$

- quand  $\alpha = \{A\}$ ,  $A$  non terminal, alors :<sup>6</sup>

$$Premier(\alpha) = \{\epsilon \text{ si } \exists A \longrightarrow \epsilon \in G\} \cup \bigcup_{\substack{A \longrightarrow \beta_1 \dots \beta_k \in G \\ \beta_1 \neq A}} Premier(\{\beta_1, \dots, \beta_k\})$$

- quand  $\alpha = \{\beta_1 \dots \beta_k\}$ , alors si  $\epsilon \notin Premier(\beta_1)$  :

$$Premier(\alpha) = Premier(\beta_1)$$

Sinon :

$$Premier(\alpha) = (Premier(\beta_1) - \{\epsilon\}) \cup Premier(\beta_2 \dots \beta_k)$$

Il est ainsi facile de déduire de cette liste de cas une implantation en fonctionnel. La fonction renvoie une liste de symboles terminaux.

```

1 let rec premier ls g = match ls with
2   | [] -> [Epsilon]
3   (* s terminal *)
4   | [s] when List.mem s (terminaux g) -> [s]
5   (* s non-terminal *)
6   | [s] ->
7     (
8       if List.mem (s, [Epsilon]) (que_faire_avec g s)
9       then [Epsilon] else []
10      ) @ (List.concat (
11        List.map (
12          function (snt, lt) ->
13            if List.length lt > 0 && List.nth lt 0 = s
14            then []
15            else premier lt g
16          ) (que_faire_avec g s)
17        )
18      )
  (* Premier(Y1Y2Y3...Yk) = ... *)

```

6. La condition éliminant les récursivités à gauche ( $\beta_1 \neq A$ ) est nécessaire pour l'implémentation, car si elle n'était pas là alors s'il existe  $A \longrightarrow A\beta \in G$  et  $A \longrightarrow \epsilon \in G$ , on aurait  $Premier(\{A\}) \supset Premier(\{A\})$ , ce qui est fâcheux.

```

19 | t::q ->
20 |   let p = premier [t] g in
21 |   if List.mem Epsilon p then
22 |     (subtract p [Epsilon])
23 |     @ (premier q g)
24 |   else
25 |     p ; ;

```

### b Construction des ensembles *Suivant*

Nos sources habituelles fournissent des définitions équivalentes mais différentes. Nous reformulerons encore la définition pour qu'en découle naturellement l'algorithme utilisé. Ainsi, on déterminera  $Suivant(B)$  de la manière suivante :

$$Suivant(B) = \{\$ \text{ si } B = \text{axiome}(G)\} \cup \bigcup_{A \rightarrow \alpha B \beta} (\text{si } \epsilon \in Premier(\beta), Suivant(A) \text{ sinon } Premier(\beta) - \{\epsilon\})$$

Ainsi, on construit  $Suivant(B)$  en lui ajoutant l'élément \$ si  $B$  est l'axiome de la grammaire, puis en itérant sur les productions de la grammaire dont la partie droite contenant  $B$  de cette façon : pour une production  $A \rightarrow \alpha B \beta$ , si  $\beta$  est une chaîne vide ou si  $Premier(\beta)$  contient  $\epsilon$ <sup>7</sup>, alors on rajoute  $Suivant(A)$  à  $Suivant(B)$ , et sinon on lui rajoute  $Premier(\beta)$  amputé de  $\epsilon$  s'il le contient.

Nous prendrons bien garde à éviter de boucler infiniment sur les récursions dans l'algorithme. Si on prend comme exemple la grammaire suivante :

$$\begin{aligned}
S &\longrightarrow A \\
A &\longrightarrow aB \\
B &\longrightarrow bA \\
B &\longrightarrow b
\end{aligned}$$

On va construire  $Suivant(B)$ . L'ensemble des productions dans lequel le non-terminal  $B$  apparaît en partie droite est  $\{A \rightarrow aB\}$ , ainsi on a  $Suivant(B) = Suivant(A)$ . Construisons donc  $Suivant(A)$  en regardant l'ensemble des productions dans lequel  $A$  apparaît en partie droite, c'est-à-dire  $\{B \rightarrow bA\}$ , et ainsi on a donc  $Suivant(A) = Suivant(B)$ . On a donc besoin de construire  $Suivant(B)$  : on va construire  $Suivant(B)$ . L'ensemble des productions dans lequel le non-terminal  $B$  apparaît en partie droite est  $\{A \rightarrow aB\}$ , ainsi on a  $Suivant(B) = Suivant(A)$ . Construisons donc  $Suivant(A)$  en regardant l'ensemble des productions dans lequel  $A$  apparaît en partie droite, c'est-à-dire  $\{B \rightarrow bA\}$ , et ainsi on a donc  $Suivant(A) = Suivant(B)$ . On a donc besoin de construire  $Suivant(B)$  : on va construire  $Suivant(B)$ . L'ensemble des productions dans lequel le non-terminal  $B$  apparaît en partie droite est  $\{A \rightarrow aB\}$ , ainsi on a  $Suivant(B) = Suivant(A)$ . Construisons donc  $Suivant(A)$  en regardant l'ensemble des productions dans lequel  $A$  apparaît en partie droite, c'est-à-dire  $\{B \rightarrow bA\}$ , et ainsi on a donc  $Suivant(A) = Suivant(B)$ . On a donc besoin de construire  $Suivant(B)$  et de toute façon ce texte est tellement petit que personne ne le lira. **Ainsi pour éviter un comportement de ce style il faudra vérifier que lorsqu'on cherche à connaître**

7. Cette condition contient la précédente, étant donné que  $Premier(\emptyset) = \{\epsilon\}$ .

$Suivant(A)$  afin de connaître  $Suivant(B)$ , on ignore les règles dont la partie gauche est  $B$ . De même, si on cherche à connaître  $Suivant(A)$  afin de connaître  $Suivant(B)$  afin de connaître  $Suivant(C)$  alors que  $Suivant(A)$  nécessite de connaître  $Suivant(C)$ , le problème pourrait se produire de façon indirecte.

Il est maintenant très simple d'écrire la fonction implantant le calcul de  $Suivant(B)$  pour un non-terminal  $B$ . Cette fonction renvoie une liste de symboles terminaux.

```

1 let rec suivant nonterm g =
2 let rec suivant_r nonterm g cf =
3   (if nonterm = axiome g then [Dollar] else [])
4   @
5   enlever_doublons (
6     List.concat (List.map (
7       (function ((np,lp),n,lg,ld) ->
8         let fs = premier ld g in
9         if ld = [] && nonterm != np && not (List.mem np cf)
10        then suivant_r np g (nonterm::cf)
11        else if List.mem Epsilon fs && not (List.mem np cf)
12        then suivant_r np g (nonterm::cf)
13        else subtract fs [Epsilon]
14      )
15    ) (ou_est nonterm g)
16  )) in suivant_r nonterm g [];;

```

## 2 Généralités sur l'analyse LR

Les faits énoncés dans cette sous-section valent pour tous les analyseurs de la famille LR. Nous y présentons quelques notions nécessaires à la construction des tables d'analyse.

### a Grammaire augmentée

L'augmentation d'une grammaire est nécessaire pour que la construction des tables d'analyse soit correcte. Comme dit précédemment, augmenter une grammaire  $G$  où l'axiome est  $S$  revient à donner la grammaire augmentée  $G' = G \cup \{S' \rightarrow S\}$ . La grammaire est accompagnée de deux opérations *Fermeture* et *Transition* qui seront détaillées plus loin.

### b Items

**Premier contact** Les tables d'analyse permettent de piloter un automate fini déterministe et déterminent sa fonction de transition. Afin de construire ces tables il est nécessaire de connaître quelle configuration correspond à quel état de l'automate. Pour cela les grands anciens ont introduit la notion d'item<sup>8</sup>. Un item désigne la configuration actuelle de l'analyseur concernant une production particulière.

8. Peu importe quand les anciens ont inventé ces items, le fait est qu'ils ne servent pas que pour l'analyse LR mais aussi pour l'algorithme d'Earley, par exemple!

Un **item LR(0)** est la donnée d'une production de la grammaire et d'une position dans la partie droite de cette production. Afin de se représenter aisément la position, on la marque par un point dans la partie droite. Ainsi l'item représentant le fait que l'analyseur a lu les trois premiers symboles de la partie droite de la production  $A \rightarrow \text{POUET}$  s'écrit  $A \rightarrow \text{POU} \bullet \text{ET}$ . Par exemple, la production  $A \rightarrow \text{BCD}$  peut fournir les quatre items suivants :

$$\begin{aligned} A &\rightarrow \bullet \text{BCD} \\ A &\rightarrow \text{B} \bullet \text{CD} \\ A &\rightarrow \text{BC} \bullet \text{D} \\ A &\rightarrow \text{BCD} \bullet \end{aligned}$$

On définit le type Objective Caml correspondant.

```
1 type item0 = production * int;;
```

Un **item LR(1)** possède une information supplémentaire ayant pour effet d'augmenter le nombre d'états de l'automate : il s'agit d'un symbole terminal qui représente le prochain symbole à lire en entrée. Ce symbole, appelé symbole de pré-vision, n'est donc utile que dans les cas où il est nécessaire de connaître le prochain symbole pour se décider. On notera les items LR(1) de cette façon :  $[A \rightarrow \text{POU} \bullet \text{ET}, b]$  avec  $b$  le symbole de pré-vision.

On définit le type Objective Caml correspondant.

```
1 type item1 = item0 * symbole;;
```

**Item et implémentation** Nous créons un type somme pour que les fonctions traitant des items de façon indépendante de leur nature LR(0) ou LR(1) puissent obtenir toutes les informations nécessaires.

```
1 type item = I0 of item0 | I1 of item1;;
```

**Opérations de base** Étant donnée la structure d'un item, il est essentiel de pouvoir proposer au moins l'opération qui consiste à obtenir le prochain symbole, c'est-à-dire intuitivement celui qui est à droite du point dans l'item. Par convention, si le point est à l'extrême droite de l'item, alors ce qui reste après est  $\epsilon$ .

```
1 let prochain_symbole it = match it with
2   | I0((nom, l), n) | I1((nom, l), n), _ ->
3     try List.nth l n with _ -> Epsilon;;
```

Une autre opération pouvant être définie permet de récupérer ce qui se trouve après le prochain symbole d'un item donné. Soit un item  $A \rightarrow \alpha \bullet B\beta$ , alors l'opération suivante permettra de récupérer la chaîne  $\beta$ , tandis que la précédente récupérait  $B$ .

```
1 let apres_prochain_symbole it = match it with
2   I0((nom, l), n) | I1((nom, l), n), _ ->
3     sublist l (n+1) (List.length l);;
```

On aura surtout besoin de deux opérations importantes afin d'exploiter tous ces items.

**Opération Fermeture** On définit la fermeture d'un ensemble d'items LR(0)  $I$ , pour une grammaire  $G$ , par  $Fermeture(I) = I \cup \{B \rightarrow \bullet\gamma / A \rightarrow \alpha \bullet B\beta \in Fermeture(I) \text{ et } B \rightarrow \gamma \in G\}$ . Pour un ensemble d'items LR(1), cette définition s'étend à  $Fermeture(I) = I \cup \{[B \rightarrow \bullet\gamma, s] / [A \rightarrow \alpha \bullet B\beta, a] \in Fermeture(I) \text{ et } B \rightarrow \gamma \in G \text{ et } s \in Premier(\beta a)\}$ .

L'algorithme permettant de construire la fermeture d'un ensemble d'items se répète jusqu'à ce qu'aucun nouvel item ne soit dans l'ensemble. Si dans le monde magique des mathématiques, rajouter des éléments à un ensemble n'agrandit pas forcément cet ensemble (penser au cas où l'élément était déjà dans l'ensemble), le monde réel est plus pragmatique et il nous faudra faire attention à traiter une liste comme un ensemble. Nous enlevons donc les doublons de la liste lorsque c'est nécessaire, ce qui est une solution pas très optimisée mais très simple et qui a l'avantage de ne pas embrouiller l'algorithme avec des détails. Le Dragon présente l'algorithme suivant. Au début, on ajoute  $I$  à  $Fermeture(I)$ , et ensuite on répète la règle suivante, affectueusement nommée la *règle 2*, jusqu'à ce que  $Fermeture(I)$  ne grandisse plus : pour chaque item<sup>9</sup>  $A \rightarrow \alpha \bullet B\beta$  présent dans l'ensemble en cours de construction, et pour chaque production<sup>10</sup>  $B \rightarrow \gamma$  ayant  $B$  comme partie gauche dans  $G$ , alors on rajoute l'item  $B \rightarrow \bullet\gamma$  dans  $Fermeture(I)$ . Lorsqu'on travaille avec des items LR(1), alors on n'oubliera pas d'itérer une troisième fois, afin que la règle 2 soit : pour chaque item  $[A \rightarrow \alpha \bullet B\beta, a]$  présent dans l'ensemble en cours de construction, et pour chaque production  $B \rightarrow \gamma$  ayant  $B$  comme partie gauche dans  $G$ , et pour chaque symbole  $s \in Premier(\beta a)$ , alors on rajoute l'item  $[B \rightarrow \bullet\gamma, s]$  dans  $Fermeture(I)$ .

**Remplissage jusqu'à ce que mort s'ensuive** L'opération qui consiste à répéter l'application d'une règle pour remplir un ensemble jusqu'à ce que l'ensemble ne grandisse plus sera nommée *remplir jusqu'à la mort* et sera encapsulée dans une fonction spéciale.

L'opération est définie récursivement par l'algorithme de la figure 2.3.

9. Cette première itération sera faite par récursion.

10. Cette seconde itération sera réalisée grâce à l'opération map sur les listes.

```

fonction RemplirJusquaLaMort(f, E, p);
début
   $J = E \cup f(E, p)$ ;
  si  $E = J$  alors
    retourner E
  sinon
    retourner RemplirJusquaLaMort(f,J,p)
fin

```

FIGURE 2.3 – Algorithme de remplissage jusqu'à ce que mort s'ensuive

Ainsi on déduit naturellement le code de la fonction implantant l'algorithme :

```

1 let rec remplir_jusqua_la_mort f l g =
2   let j = enlever_doublons (l @ (f l g)) in
3   if l = j then l else remplir_jusqua_la_mort f j g;;

```

**Fermeture d'un ensemble d'items** Nous pouvons maintenant appliquer l'algorithme du Dragon implanté dans le style fonctionnel. Voici comment nous avons écrit l'opération *Fermeture* avec des items LR(0).

```

1 let rec fermeture0 l g =
2   let rec regle2 l g = match l with
3     | [] -> []
4     | t::q -> (
5       List.map (function prod -> I0(prod, 0))
6         (que_faire_avec g (prochain_symbole t))
7     ) @ (regle2 q g)
8 in enlever_doublons (l @ (remplir_jusqua_la_mort regle2 l g));;

```

L'opération *Fermeture* avec des items LR(1) est écrite ci-dessous.

```

1 let rec fermeture1 l g =
2   let rec regle2 l g = match l with
3     | [] -> []
4     | t::q -> List.flatten (
5       List.map
6         (function prod ->
7           match t with I0(x) ->
8             failwith "4è_dimension_inside"
9           | I1(i0,s) ->
10            List.map (function x ->
11              I1((prod,0),x)
12            )
13            (premier (s::(apres_prochain_symbole t)
14              ) g)
15          ) (que_faire_avec g (prochain_symbole t))
16     ) @ (regle2 q g)
16 in

```

```
17 enlever_doublons (l @ (remplir_jusqua_la_mort regle2 l g));;
```

Afin de proposer une fonction *Fermeture* qui fonctionne indépendamment du type d'item, nous écrivons une fonction qui redirige l'appel selon un motif particulier.

```
1 let fermeture l g = match l with
2   | [] -> []
3   | IO(x)::q -> fermeture0 l g
4   | I1(x)::q -> fermeture1 l g
5 ;;
```

**Opération Transition** On définit la transition d'un ensemble d'items  $I$  par  $Transition(I, X) = Fermeture(J)$  avec  $J = \{A \rightarrow \alpha X \bullet \beta / A \rightarrow \alpha \bullet X \beta \in I\}$ .

On a juste à déterminer l'ensemble  $J$  en parcourant les éléments de  $I$ , en considérant uniquement les items dont le prochain symbole est  $X$  et en faisant passer le point après le  $X$ .

```
1 let transition l s g =
2   let rec calcul_ensemble l s g = match l with
3     | [] -> []
4     | IO(p,n)::q when prochain_symbole (IO(p,n)) = s ->
5       IO(p, n+1)::(calcul_ensemble q s g)
6     | I1((p,n),z)::q when prochain_symbole (I1((p,n),z)) = s ->
7       I1((p, n+1),z)::(calcul_ensemble q s g)
8     | _::q -> calcul_ensemble q s g
9   in fermeture (calcul_ensemble l s g) g ;;
```

### c Construction des ensembles d'items canoniques

Chaque état de l'automate est associé à un ensemble d'items. Cela signifie qu'un ensemble d'items indique bien un état possible de l'automate par des conditions sur des productions, qui sont les items eux-mêmes. La création des ensembles d'items canoniques est donc ce qui va déterminer les états de l'automate.

Nous faisons confiance au Dragon dont nous tirons l'algorithme qui permet de construire  $Items(G')$ , avec  $G'$  une grammaire augmentée, qui est présenté à la figure 2.4.

L'implémentation de l'algorithme est proche celui de la fonction *Fermeture*, mais nous utilisons l'opération `fold_left` sur les listes au lieu de l'opération `map`, afin de pouvoir reconstruire la liste en filtrant les éléments<sup>11</sup>. On crée donc la fonction `itemsLR` en fonction du nombre de symboles de pré-vision (limité à 0 ou 1) et d'une grammaire supposée augmentée.

```
1 let itemsLR k g =
2   let rec fonction_quelconque l g = match l with
3     | [] -> []
4     | t::q -> (
5       List.fold_left (
6         fun l x ->
```

11. Il existe bien d'autres manières de le faire.

```

fonction Items( $G'$ );
début
   $C = \{Fermeture([S' \rightarrow \bullet S])\}$ a
  RemplirJusquaLaMort(l'ensemble  $C$  avec la règle début
    pour chaque  $I \in C$  faire
      pour chaque symbole  $X \in G'$  faire
        si  $Transition(I, X) \neq \emptyset$  alors  $C = C \cup \{Transition(I, X)\}$ 
      finfaire
    finfaire
  fin)
retourner  $C$ 
fin

```

a. Pour construire la collection des ensembles d'items LR(1) on fera plutôt  $C = \{Fermeture([S' \rightarrow \bullet S, \$])\}$

FIGURE 2.4 – Algorithme de construction de la collection canonique des ensembles d'items

```

7           let ttx = transition t x g in
8             if List.length ttx > 0
9             then l @ [ttx] else l
10          ) [] (symboles g)
11          ) @ (fonction_quelconque q g)
12 in
13 let f = match k with
14   | 0 -> fermeture [I0((List.nth g 0),0)] g
15   | 1 -> fermeture [I1(((List.nth g 0),0),Dollar)] g
16   | _ -> failwith "Je_ne_suis_pas_Jésus,_non_plus_!"
17 in enlever_doublons
18   ([f] @ (remplir_jusqua_la_mort fonction_quelconque [f] g))
19 ;;

```

On remarquera qu'avec cet algorithme, l'état initial de l'analyseur sera toujours celui qui a été construit à partir de l'ensemble contenant l'item relatif à l'axiome de la grammaire augmentée, en première position de partie droite, et avec le symbole de pré-vision \$ si symbole de pré-vision il y a.

### 3 Construction des tables d'analyse

Soit  $E$  l'ensemble des états de l'automate à pile représentant l'analyseur. La construction des tables d'analyse pour un grammaire  $G$  d'un type donné revient à précalculer les valeurs des fonctions  $Action : (E \times \Sigma) \mapsto \mathcal{A}$  et  $Successeur : (E \times V) \mapsto E$  dans des tables de correspondance. Nous construisons ces tables en Objective Caml comme une liste de paires  $(e, s), a$  ou  $(e, A), b$  avec  $e, b \in E$  un état,  $s \in \Sigma$  un symbole terminal,  $A \in V$  un symbole non-terminal et  $a \in \mathcal{A}$  une action. L'ensemble  $\mathcal{A}$  contient l'ensemble des actions possibles, à savoir :  $\mathcal{A} = \{Accepter, Erreur\} \cup \{Réduire(p) / p \in G\} \cup \{Décaler(e) / e \in E\}$ . La liste est utilisée comme une table de correspondance grâce à l'opération de liste assoc.

Nous ferons une petite entrave aux langages fonctionnels en utilisant une variable dont la valeur peut être modifiée afin de construire les tables.

### a Analyse SLR(1)

L'analyse SLR repose sur la construction des tables d'analyse à partir de la collection d'ensembles d'items LR(0) canonique pour une grammaire. Il en résulte que ce type d'analyse reconnaît moins de grammaires que si les tables étaient construites à partir d'items LR(1). En effet, les cas où le symbole de pré-vision est utile pour opérer une décision sont ici une cause de conflits *décaler/réduire* ou *réduire/réduire*. Cependant, les tables SLR ne sont pas construites sans aucune information pertinente. En effet, nous utiliserons les ensembles *Suivant* dans la construction des tables. On parle de SLR(1) car les ensembles *Suivant* sont basés sur les ensembles *Premier* qui associent à un terminal l'ensemble composé uniquement de ce même terminal<sup>[4]</sup>. Pour le cas SLR( $k$ ) avec  $k > 1$  nous aurions été obligés d'utiliser une définition bien moins simple.

L'idée centrale de la construction des tables SLR est de réduire par  $X \rightarrow \alpha$  pour l'état  $i$  et le symbole d'entrée  $s$  si et seulement si  $X \rightarrow \alpha \in I_i$  et  $s \in \text{Suivant}(X)$ . Une autre idée importante, qui est appliquée également dans les autres types d'analyseurs, est qu'il est primordial de détecter d'éventuelles actions conflictuelles. Une grammaire est d'un type particulier lorsque l'analyseur du même type réussit à construire ses tables d'analyse. Dans le cas où la construction est impossible, c'est que des conflits *décaler/réduire* ou des conflits *réduire/réduire* sont apparus. De tels conflits sont détectés lorsque l'analyseur veut ajouter une entrée à la table *Action* pour un état  $e$  et un symbole  $s$  alors qu'il existe déjà une telle entrée.

On va donc détailler l'algorithme de construction des tables. On commence par construire la collection  $C = \{I_0, \dots, I_n\}$  canonique d'ensembles d'items LR(0) pour définir les états de l'automate. Une fois cette étape faite, on peut déterminer, pour chaque état  $i$  de l'analyseur construit à partir de l'ensemble  $I_i$ , les actions d'analyse, et ce en examinant chaque item  $I \in I_i$  :

- si  $I = A \rightarrow \alpha \bullet \beta$ , alors on calcule  $J = \text{Transition}(I_i, a)$ , puis si  $\exists j / I_j = J$  et que  $\text{Action}(i, a)$  n'est pas encore rempli, alors  $\text{Action}(i, a) = \text{Décaler}(j)$ . Si  $\text{Action}(i, a)$  était déjà rempli par une action  $\text{Réduire}(x)$ ,  $x \in G$ , alors notifier d'un conflit *décaler/réduire* ;
- si  $I = A \rightarrow \alpha \bullet$ , alors calculer  $J = \text{Suivant}(A)$  et pour tous les terminaux  $a \in J$ , si  $A \neq S'$  et si  $\text{Action}(i, a)$  n'est pas encore rempli, alors  $\text{Action}(i, a) = \text{Réduire}(A \rightarrow \alpha)$ . Si  $\text{Action}(i, a)$  était déjà rempli par une action  $\text{Réduire}(x)$ ,  $x \in G$ , here-document déjà rempli par une action  $\text{Décaler}(e)$ ,  $e \in E$ , alors notifier d'un conflit *décaler/réduire*, ou s'il était déjà rempli par une autre action, alors déclarer cet analyseur apte à voyager dans le temps ;
- si  $I = S' \rightarrow S \bullet$ , alors  $\text{Action}(i, \$) = \text{Accepter}$ .

Ensuite, il faut remplir la table *Successeur* comme pour la méthode suivante (*i.e.* copier-coller de **b**). Pour chaque état  $i$ , et pour chaque non-terminal  $A$ , s'il existe  $j$  tel que  $\text{Transition}(I_i, A) = I_j$  alors  $\text{Successeur}(i, A) = j$ .

Si les tables sont construites, alors c'est dans la poche, Gavroche. L'implantation des algorithmes de création de tables étant dans un style pas très orthodoxe et faisant appel aux

traits impurs d'Objective Caml, nous ne salirons pas le rapport avec leur contenu. Pour se faire du mal, jetons un coup d'œil à des extraits<sup>12 13 14</sup>.

```

1 ...
2 for i = 0 to List.length canonique - 1 do
3 ...
4 | _ -> failwith "Nous_tombons_dans_la_4e_dimension."
5 ...
6 actions := (!actions @ [(i,a), DecalerEtEmpilerEtat j])

```

### b Analyse LR(1)

Contrairement à l'analyse SLR, l'analyse LR(1) construit les tables d'analyse à partir des items LR(1) comportant donc une information de pré-vision. Le nombre d'états croît donc de manière relativement fabuleuse. Si l'analyse LR( $k$ ) est généralement présentée après l'analyse SLR, c'est principalement pour des raisons didactiques, mais l'Histoire veut que l'analyse LR( $k$ ) ait été inventée avant<sup>[5]</sup>. Les différentes manières d'exprimer le processus d'analyse LR( $k$ ) qui ont été trouvées au fil du temps ont permis de considérer la méthode LR(1) comme une *amélioration* de la méthode SLR car on utilise presque les mêmes choses mais on rajoute de l'information, en la présence du symbole de pré-vision pour les items.

Les items LR(1) étant déjà construits, il ne reste plus qu'à commenter l'algorithme de construction des tables d'analyse LR(1). Pour chaque état  $i$  de l'analyseur, construit à partir de l'ensemble  $I_i$ , on détermine les actions d'analyse en examinant chaque item  $I \in I_i$  :

- si  $I = [A \rightarrow \alpha \bullet a\beta, b]$ , alors on calcule  $J = \text{Transition}(I_i, a)$ , puis si  $\exists j / I_j = J$  et que  $\text{Action}(i, a)$  n'est pas encore rempli, alors  $\text{Action}(i, a) = \text{Décaler}(j)$ . Si  $\text{Action}(i, a)$  était déjà rempli par une action  $\text{Réduire}(x)$ ,  $x \in G$ , alors notifier d'un conflit *décaler/réduire* ;
- si  $I = [A \rightarrow \alpha \bullet, a]$  avec  $A \neq S'$  et que  $\text{Action}(i, a)$  n'est pas encore rempli, alors  $\text{Action}(i, a) = \text{Réduire}(A \rightarrow \alpha)$ . Si  $\text{Action}(i, a)$  était déjà rempli par une action  $\text{Réduire}(x)$ ,  $x \in G$ , alors notifier d'un conflit *réduire/réduire*, ou s'il était déjà rempli par une action  $\text{Décaler}(e)$ ,  $e \in E$ , alors notifier d'un conflit *décaler/réduire*, ou s'il était déjà rempli par une autre action, alors déclarer cet analyseur bon pour l'asile de fous ;
- si  $I = [S' \rightarrow S \bullet, \$]$ , alors  $\text{Action}(i, \$) = \text{Accepter}$ .

Ensuite, il faut remplir la table *Successeur* comme précédemment (*i.e.* copier-coller de **a**). Pour chaque état  $i$ , et pour chaque non-terminal  $A$ , s'il existe  $j$  tel que  $\text{Transition}(I_i, A) = I_j$  alors  $\text{Successeur}(i, A) = j$ . Si les tables sont construites, alors bravo, c'est gagné.

### c Analyse LALR(1)

Nous avons déjà évoqué le problème des tables LR(1) : elles comportent un trop grand nombre d'états pour des jeunes hommes des années 70 aux chemises bariolées et aux cheveux gras. La méthode LALR permet de produire des tables d'analyse de taille équivalente aux tables SLR, et ce à partir des tables LR(1). Le nombre de grammaires reconnues est moindre par rapport à l'analyse LR(1) mais il reste nettement plus grand que pour l'analyse SLR. C'est

12. Ventrebleu !

13. "La quatrième dimension ? C'est quand même fou de tomber dans la quatrième dimension." — Julien

14. Ouch !

pour cette raison que l'analyse LALR est couramment utilisée dans le monde réel. La clé de la méthode LALR repose sur la factorisation de la collection d'items canonique LR(1) par rapport aux cœurs des ensembles d'items.

Soit l'ensemble d'items  $I_1 = \{[R_1, s_1], \dots, [R_k, s_k]\}$ , alors le **cœur** de  $I_1$  est  $\{R_1, \dots, R_k\}$ . Soit maintenant  $I_2 = \{[R_1, s'_1], \dots, [R_k, s'_k]\}$ , alors  $I_1$  et  $I_2$  ont le même cœur. Lors de la construction des tables d'analyse LALR, si la collection d'items LR(1) obtenue comporte  $I_1$  et  $I_2$ , alors on factorisera ces deux ensembles en  $I_{12} = \bigcup_{p=1}^k \{[R_1, s_p], \dots, [R_k, s_p], [R_1, s'_p], \dots, [R_k, s'_p]\}$ . Le même comportement est à prévoir lorsqu'il existe plus d'un symbole  $s$  pour lequel  $[R_i, s] \in I_n$ , mais on n'a pas pris en compte le cas ici pour simplifier les choses.

Néanmoins, lorsqu'on factorise les cœurs des ensembles d'items canoniques, on s'expose à créer des conflits *réduire/réduire*<sup>15</sup>. Pour reprendre brièvement l'exemple du Dragon<sup>[1]</sup>, soit une grammaire augmentée LR(1) suivante :

$$\begin{aligned} S' &\longrightarrow S \\ S &\longrightarrow aAd|bBd|aBe|bAe \\ A &\longrightarrow c \\ B &\longrightarrow c \end{aligned}$$

On construit la collection d'ensembles d'items et on trouve notamment l'ensemble  $I_i = \{[A \longrightarrow c\bullet, d], [B \longrightarrow c\bullet, e]\}$  et l'ensemble  $I_j = \{[A \longrightarrow c\bullet, e], [B \longrightarrow c\bullet, d]\}$  que l'on fusionnera en un ensemble  $I_{ij} = \{[A \longrightarrow c\bullet, d], [A \longrightarrow c\bullet, e], [B \longrightarrow c\bullet, d], [B \longrightarrow c\bullet, e]\}$ . Lors de la construction des tables d'analyse LALR, nous trouverons que  $[A \longrightarrow c\bullet, e] \in I_{ij}$  et nous pourrions dire que *Action*( $ij, e$ ) donne "réduire par  $A \longrightarrow c$ ", puis nous trouverons que  $[B \longrightarrow c\bullet, e] \in I_{ij}$  et nous voudrions dire que *Action*( $ij, e$ ) donne "réduire par  $B \longrightarrow c$ ", mais la place est déjà prise, ce qui cause un échec de la construction des tables en raison d'un conflit *réduire/réduire*.

**Factorisation des états semblables** Nous devons construire un algorithme capable de traquer les ensembles ayant le même cœur et de les factoriser. Soit  $C = \{I_0, \dots, I_n\}$ , on veut construire  $C' = \{J_0, \dots, J_m\}$ . Pour chaque ensemble  $I_i$ , s'il existe  $k$  tel que  $I_i$  a le même cœur que  $J_k$ , alors  $J_k = J_k \cup I_i$ , et sinon si  $\text{card}(C') = p$  alors  $J_{p+1} = I_i$  et  $C' = \{J_0, \dots, J_p\} \cup J_{p+1}$ . Au final, l'ensemble  $C'$  sera construit.

**Remarque sur fonction *Transition*** Le Dragon précise une autre manière de voir la fonction *Transition* en s'appuyant sur la remarque suivante : si  $J$  est ce que nous appelons la factorisation des ensembles d'états  $I_1$  à  $I_m$  alors les cœurs de *Transition*( $I_i, X$ ) et de *Transition*( $I_j, X$ ) sont identiques, et ce pour  $1 < i < j < m$ . Ainsi *Transition*( $J, X$ ) est la factorisation de tous les ensembles d'items qui ont le même cœur que *Transition*( $I_j, X$ ) pour un  $1 < j < m$ . Nous en resterons pour ce projet à l'algorithme vu précédemment pour éviter de complexifier encore un programme déjà lourd.

<sup>15</sup>. On ne peut pas créer de conflits *décaler/réduire* avec une telle factorisation car la décision de décaler ne dépend pas du symbole de pré-vision.

---

**Construction des tables d'analyse** Les tables sont construites à partir de  $C'$ , mais de la même manière que pour un analyseur LR(1).

# CONCLUSION

En choisissant un sujet sur les compilateurs, et plus particulièrement l'analyse syntaxique, nous sommes tombés sur un monde très vaste. Des linguistes, en passant par les philosophes pour arriver aux informaticiens, de nombreuses personnes ont mis leur grain de sel dans cette machine. Que ce soit de petites astuces, ou des publications majeures, ou des anecdotes qui prêtent à sourire <sup>16</sup> l'histoire de la compilation pourrait nous occuper pendant longtemps.

Ce projet n'aura pas permis de percée notoire dans le monde de la compilation, mais il nous aura fait découvrir une liaison qui ne paraissait pas évidente l'année passée, en cours de langages formels et compilation. Les bases acquises dans ce module ont pourtant été fort utiles.

Pour nous lancer dans l'aventure, la bibliographie du sujet précisait une source écrite, heureusement disponible en plusieurs exemplaires à la BU. Quel ne fut pas notre étonnement de tomber sur la Bible des compilateurs <sup>17</sup> ! Après avoir tâtonné le bout de la queue du Dragon, nous avons dû trouver d'autres sources et nos recherches nous ont mené vers deux types de matériel. Premièrement, nous avons trouvé des supports pédagogiques, cours, exercices ou notes éparses rédigées par des enseignants dont la plus grande partie recopiait notre cher Dragon faute d'avoir l'autorisation d'en reproduire les pages. Heureusement, nous avons tout de même réussi à trouver des reformulations et d'autres angles de vue sur le problème lors de nos balades dans tout ce matériel scolaire. Secondement, nous avons décidé de nous plonger dans la lecture issue de la recherche. L'inconvénient apparent de ces articles, mémoires de thèse et autres shakespeareries est, mis à part leur langue commune, qu'ils ont l'air bien plus inaccessibles et poussiéreux que ces superbes cours bien présentés avec Beamer ou le plus récent des Powerpoint. Malgré cela, les apparences sont parfois trompeuses, et un avantage indéniable de la documentation scientifique est qu'il est possible de les replacer dans un contexte historique et ainsi de comprendre le cheminement des idées et des techniques. En mêlant les sources comme nous l'avons fait, nous avons pu mieux nous rendre compte de la tâche complexe dont nous nous acquittons aujourd'hui.

On constatera avec joie <sup>18</sup> que l'analyse syntaxique est un domaine toujours en activité et que les acquis du passé sont parfois remis en question car les contraintes matérielles associées à l'appréciation des qualités de certains algorithmes ont changé <sup>[7]</sup>. Et la tendance actuelle des constructeurs de matériel (toujours plus de cœurs, de nouvelles architectures processeurs. . .) couplée à la nouvelle mode des développeurs de logiciels (cloud computing,

---

16. La découverte du premier bug informatique par Grace Hopper est hilarante

17. Et quel ne fut pas notre second étonnement quand nous avons appris qu'elle n'était pas écrite de la main de Jean Pallo !

18. Et avec un peu d'empathie envers les futurs étudiants qui auront toujours des raisons d'écourter leurs nuits...

nouveau langage comme Ruby on Rails, etc.) porte vers une utilisation des ordinateurs très différente de celle d'aujourd'hui.

Dans le cadre du module Algorithmique et Complexité, nous pouvons affirmer que nous avons atteint nos objectifs pour ce qui est de l'implantation et de la recherche d'algorithmes, et de la manipulation de structures de données complexes, alors que nous estimons que le mot "complexité" est passé à la trappe. Nous avons abandonné l'usage de ce mot peu après avoir commencé à réfléchir à la façon d'écrire le projet en langage fonctionnel : les calculs de complexité pour les fonctions récursives ne sont pas notre tasse de thé, alors nous avons préféré abandonner cet aspect. En même temps, nous n'avons pas à ce jour d'idée claire sur l'influence de ce style de programmation sur la performance d'un algorithme étant données les optimisations propres à ce type de langage, mais nous savons que le style est un facteur important, les algorithmes étant trop souvent liés à un style particulier de programmation.

# ANNEXES



## SOMMAIRE

1	PRÉ-REQUIS . . . . .	36
2	UTILISATION . . . . .	37
3	EXEMPLES D'UTILISATION . . . . .	38
4	CAS AMBIGUËS . . . . .	39

**D**ANS ces annexes, vous trouverez tous les renseignements permettant l'utilisation du programme dahu.

## I Pré-requis

Avant toute chose, il faut récupérer l'archive contenant les sources et les exemples. L'archive est compressée en *bzip2*. Commençons par l'extraire :

```
1 tar xavf dahu.tar.bz2
```

Nous avons créé un *makefile* pour automatiser la compilation et l'installation de l'application.

```
1 # Makefile
2 CC=ocamlc
3 CFLAGS=-g
4 LDFLAGS=-g str.cma -cclib -lstr
5 EXEC=dahu
6 INSTALL = install -c -m 755
7 INSTALLDATA = install -c -m 644
8 prefix = /usr
9 bindir = $(prefix)/bin
10 mandir = $(prefix)/man/man1
11 all: $(EXEC)
12
13 .SUFFIXES: .ml .cmo
14
15 $(EXEC): commun.cmo analyse.cmo tables.cmo interface.cmo main.cmo
16     $(CC) $(LDFLAGS) -o $@ $^
17
18 %.cmo: %.ml
19     $(CC) -c $(CFLAGS) $^
20
21 install:
22     $(INSTALL) dahu $(bindir)
23     test -d $(mandir) || mkdir $(mandir)
24     $(INSTALLDATA) dahu.1 $(mandir)
25
26 uninstall:
27     test ! -f $(bindir)/dahu || rm -v $(bindir)/dahu
28     test ! -f $(mandir)/dahu.1 || rm -v $(mandir)/dahu.1
29
30 clean:
31     rm -rf $(EXEC) *.cmo *.cmi
```



```

21 Options
22 -----
23     -h      Affiche l'aide mémoire.
24     -a      Affiche les auteurs du programme.
25     -v      Affiche la version du programme.
26
27 Documentation
28 -----
29     Analyseur syntaxique écrit en Ocaml qui réalise une analyse LR.
30     Le programme utilise le fichier de grammaire entré en paramètre
31     pour sortir un arbre syntaxique en fonction de l'expression
32     spécifiée.
33
34     Veuillez consulter le manuel pour plus de renseignements.

```

Le manuel est disponible grâce à la commande `man dahu`.

### 3 Exemples d'utilisation

Prenons comme exemple la grammaire fournie dans le sujet :

```

1 slr
2 E : T
3 E : E PlusOuMoins T
4 T : F
5 T : T MultOuDiv F
6 F : nombre
7 F : ( E )
8 PlusOuMoins : +
9 PlusOuMoins : -
10 MultOuDiv : *
11 MultOuDiv : /

```

Remarquez la première ligne, qui sert à spécifier à DAHU le type d'analyse à effectuer. Rappelons que DAHU gère les grammaires *LR*, *SLR* et *LALR*.

Utilisons donc cette grammaire avec l'expression *nombre + nombre × nombre*. Pour visualiser directement le résultat dans une fenêtre gtk, on peut utiliser cette commande :

```

1 dahu arithmetique | dot -Tgtk
2 nombre + nombre * nombre
3 QUIT (ou CTRL+d)

```

La figure A.2 présente le résultat. Testons maintenant avec l'expression  $(\text{nombre} + \text{nombre}) \times \text{nombre}$ . La figure A.3 présente le résultat.

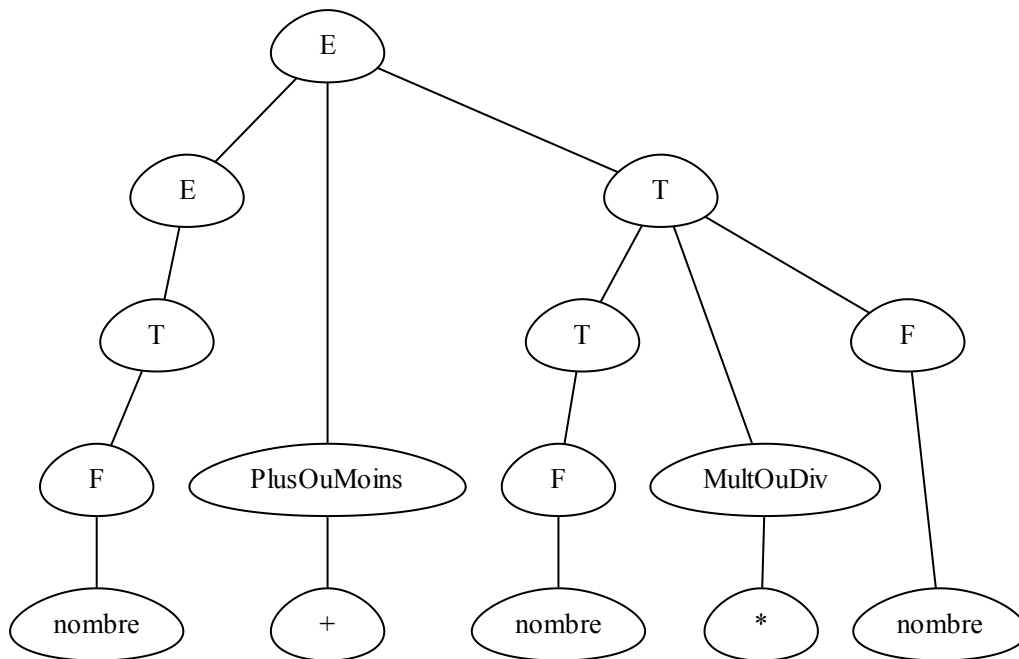


FIGURE A.2 – Arbre syntaxique pour l'expression "nombre + nombre × nombre"

Voici un exemple avec la grammaire utilisée lors de la présentation :

```

1 slr
2 E : E a
3 E : a

```

L'expression  $a a a a a$  donnera l'arbre syntaxique A.4.

## 4 Cas ambigus

Nous fournissons en exemple une grammaire *lr* ambiguë, nommée `if_then_else_ambigue`. L'utilisation de cette grammaire donnera une erreur présentée à la figure A.5 :

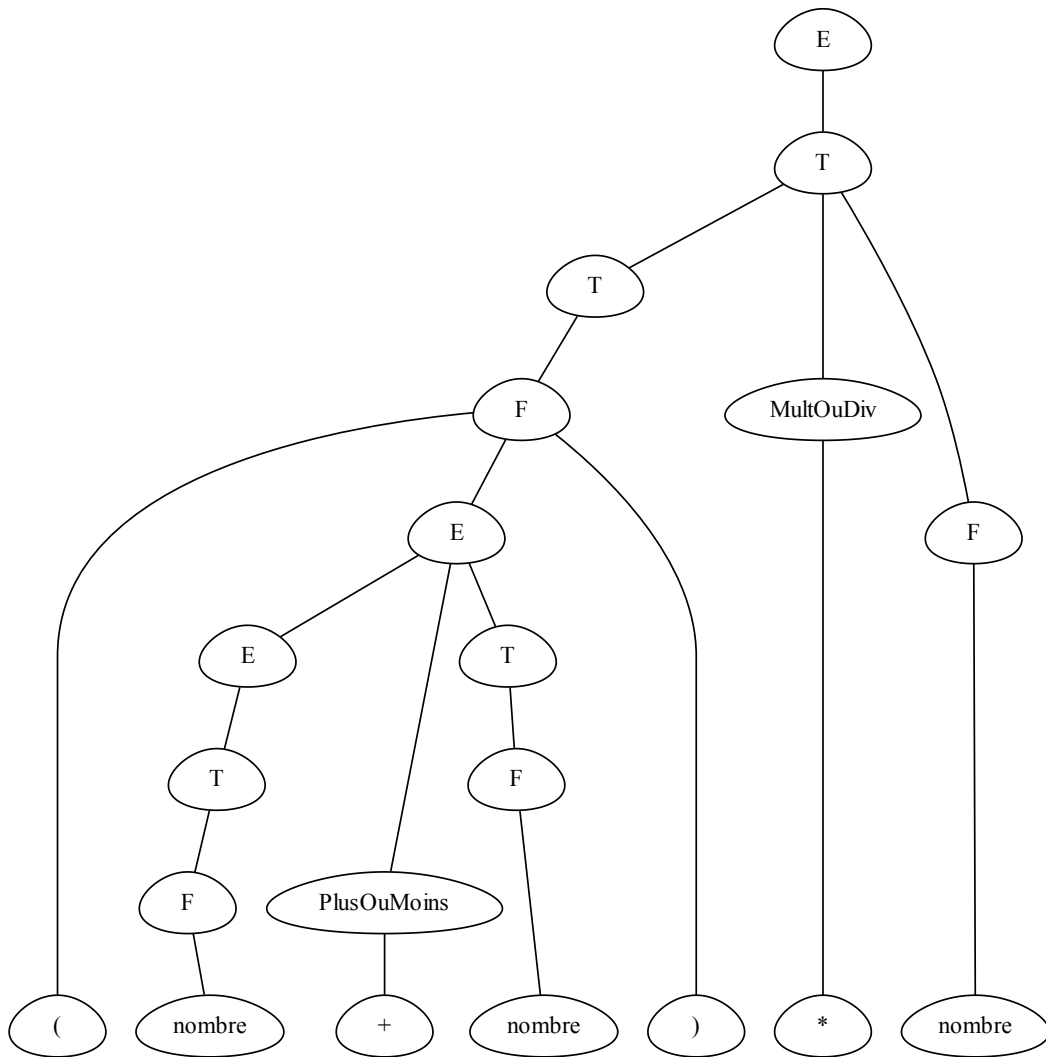


FIGURE A.3 – Arbre syntaxique pour l'expression "(nombre + nombre) × nombre"

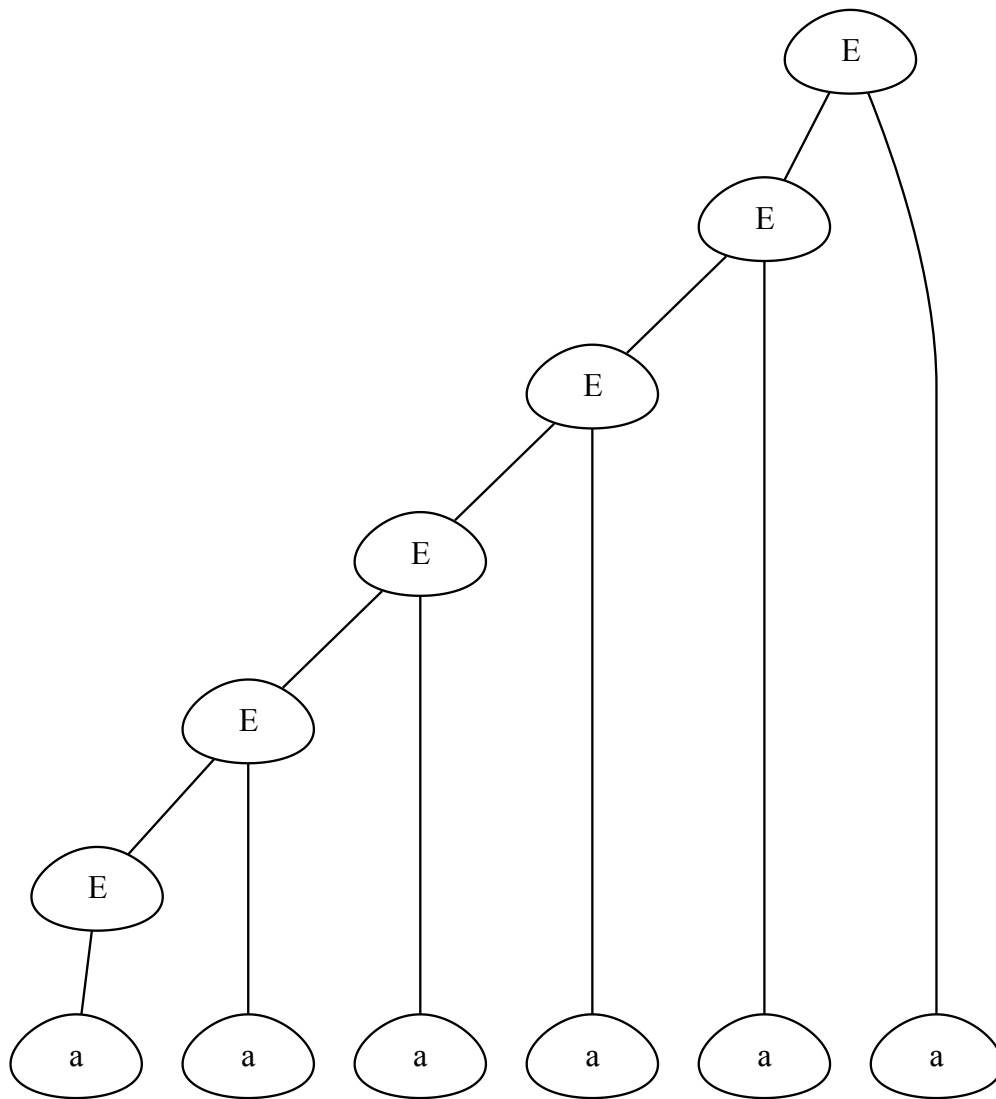


FIGURE A.4 – Arbre syntaxique pour l'expression  $a a a a a a$ .

```

-> dahu if_then_else_ambigue
if_then_else_ambigue : Conflit décaler/réduire entre décaler 'else' et réduire p
ar `stmt -> if expr then stmt`.

```

FIGURE A.5 – Conflit décaler/réduire pour la grammaire LR `if_then_else`

# BIBLIOGRAPHIE

- [1] Alfred Aho, Ravi Sethi, et Jeffrey Ullman. *Compilateurs. Principes, techniques et outils*. InterEditions, 1991. (Cité pages iii, 3, 10, 18, 20 et 31.)
- [2] J.W. Backus, R.J. Beeber, S. Best, R. Goldberg, L.M. Haibt, H.L. Herrick, R.A. Nelson, D. Sayre, P.B. Sheridan, h. Stern, I. Ziller, R.A. Hugues, et R. Nutt. The fortran automatic coding system. *Western joint computer conf*, pages 188–198, 1957. (Cité pages 3 et 4.)
- [3] James Brunskill. First and follow sets. <http://www.jambe.co.nz/UNI/FirstAndFollowSets.html>. (Cité page 20.)
- [4] Franklin Lewis DeRemer. *Practical translator for LR(k) languages*. PhD thesis, Massachusetts institute of technology, 1969. (Cité page 29.)
- [5] Donald E. Knuth. On the translation of languages from left to right. *INFORMATION AND CONTROL*, 8 :607–639, 1965. (Cité page 30.)
- [6] David Padua. The fortran I compiler. *Computing in Science and Engineering*, 2 :70–75, 2000. (Cité page 3.)
- [7] Terence Parr et Russell Quong. Ll and lr translator need k. *SIGPLAN Notices*, 31(2), 1996. (Cité pages 7 et 33.)