Aide Mémoire Ocamllex & Ocamlyacc

Armelle Bonenfant

9 novembre 2010

Résumé

Ce document est une traduction/résumé des documentations de Ocamllex et Ocamlyacc en ligne :
http ://plus.kaist.ac.kr/ shoh/ocaml/ocamllex-ocamlyacc/ocamllex-tutorial et
http ://plus.kaist.ac.kr/ shoh/ocaml/ocamllex-ocamlyacc/ocamlyacc-tutorial
(Copyright (C) 2004 SooHyoung Oh.).

Il présente les fonctionnalités de Ocamllex et de Ocamlyacc.

Table des matières

Liste des tableaux

Table des figures

Chapitre 1

Ocamllex

Ocamllex est un outil d'OCaml pour générer des *scanners* : programmes qui reconnaissent des motifs lexicaux à l'aide de *descriptions* sous forme de paire (*expression régulière, règle*). Ocamllex génère un exécutable qui effectue l'analyse lexicale et également le code Ocaml correspondant.

1.1 Commandes et structure

La commande pour lancer le lexer :

```
ocamllex program.mll
```

FIG. 1.1 – Ligne de commande

1.2 Input

Le fichier d'entrée (fig 1.2) d'Ocamllex contient 4 sections : *header, definitions, rules* et *trailer*. Son extension est `".mll"`.

Il faut retenir que :

- *header* et *rules* sont obligatoires
- *header* et *trailer* entre accolades, code Ocaml qui sera remis "tel quel" dans le fichier ocaml de sortie.
- *definitions* contient les déclarations des expressions régulières et leurs identifiant.
- *rules* contient des points d'entrée pour définir l'effet de l'analyse lexicale.

Les patterns sont décrit dans la section suivante 1.3.

1.3 Patterns

Les motifs ou *patterns* ont le format d'expressions régulières dans le style de lex à l'aide d'une syntaxe Caml-like. La table 1.1 reprend ces expressions régulières.

```
(* header section *)
{ header }

(* definitions section *)
let ident = regexp
let ...

(* rules section *)
rule entrypoint [arg1... argn] = parse
  | pattern { action }
  | ...
  | pattern { action }
and entrypoint [arg1... argn] = parse
  ...
and ...

(* trailer section *)
{ trailer }
```

FIG. 1.2 – Structure du fichier d'entrée `".mll"`

'c'	match the character 'c'
_	match any character
eof	match an end-of-file
"foo"	match the literal string "foo"
['x' 'y' 'z']	matches either an 'x', a 'y', or a 'z'
['a' 'b' 'j'–'o' 'Z']	matches an 'a', a 'b', any letter from 'j' through 'o', or a 'Z'
[^ 'A'–'Z']	a "negated character set"
[^ 'A'–'Z' 'n']	any character EXCEPT an uppercase letter or a newline
r*	zero or more r's, where r is any regular expression
r+	one or more r's, where r is any regular expression
r?	"an optional r"
ident	the expansion of the "ident"
(r)	parentheses are used to override precedence (see below)
rs	"concatenation"
r|s	either an r or an s
r#s	match the difference of the two specified character sets
r **as** ident	bind the string matched by r to identifier ident

TAB. 1.1 – Tableau des motifs

Les expressions régulières citées dans la table 1.1 sont triées selon les priorités de précédence du plus fort au plus faible : '*' et '+' ont la plus forte précédence, puis '?', 'concatenation', '|', et enfin 'as'.

Par exemple, `"foo" | "bar"*` = `("foo")|("bar"*)` = "foo" ou zero-ou-plusieurs "bar", et `("foo"|"bar")*` = zero-ou-plusieurs "foo"-ou-"bar" (ex : "foobarbarbarfoobar", "", "barbar"...)

Remarque : l'expression newline peut être reconnue par `"[^ 'A'-'Z']"` à moins que `"n"` soit explicitement présent dans la négation : `"[^ 'A'-'Z' '\n']"` (non standard).

1.4 Match

Le principe est le suivant : si il y a plusieurs patterns qui correspondent à l'expression saisie, le principe du "plus long match" est appliqué. C'est-à-dire qu'entre les patterns, "ding", "dong" et "dingdong", l'expression saisie sera associée au pattern "dingdong". S'il y a plusieurs patterns de longueur égale, on prend le premier de la liste.

Lorsque le motif correspondant est trouvé, le texte correspondant, appelé *token* est considéré comme disponible sous la forme d'une chaîne de caractères. L'action correspondante au motif filtré est alors exécuté (voir description des actions dans la section 1.5) et le reste de l'input est scanné pour un motif suivant.

Si aucune correspondance n'est trouvée, le scanner lève l'exception `"lexing : empty token"`.

Voici quelques exemples qui montrent le fonctionnement du filtrage.

Les deux premiers cas (1.3,1.4) illustrent que lorsque deux motifs peuvent être reconnus, Ocamllex choisit le premier de la liste. Ainsi, dans le deuxième cas (1.4), le pattern "ding" est donc inutile.

```
(* rules section *)
rule token = parse
  | "ding"      { print_endline "Ding" }                (* "ding"
    pattern *)
  | ['a'-'z']+ as word                                  (* "word"
    pattern *)
                { print_endline ("Word: " ^ word) }
  ...
```

FIG. 1.3 – Importance de l'ordre (1)

```
(* rules section *)
rule token = parse
  | ['a'-'z']+ as word                          (* "word" pattern *)
                { print_endline ("Word: " ^ word) }
  | "ding"      { print_endline "Ding" }                (* "ding"
    pattern *)
  | ...
```

FIG. 1.4 – Importance de l'ordre (2)

Dans ce troisième exemple (1.5), on choisit le plus long motif reconnu. Il y a trois patterns : ding, dong et dingdong.

```
(* rules section *)
rule token = parse
  | "ding"      { print_endline "Ding" }                (* "ding"
    pattern *)
  | "dong"      { print_endline "Dong" }                (* "dong"
    pattern *)
  | "dingdong"  { print_endline "Ding-Dong" }   (* "dingdong" pattern
     *)
  ...
```

FIG. 1.5 – Plus long motif reconnu

Lorsque "dingdong" est donné en input, deux choix sont possibles : soit les motifs ding + dong soit le motif dingdong. Par le principe du plus long motif reconnu, le motif dingdong est choisi.

Il est possible d'appliquer le principe du plus court motif reconnu en remplaçant le mot clé parse par shorstest . Le principe du premier de la liste est conservé.

1.5 Action

Chaque motif dans une règle a une action correspondante, qui est une expression Ocaml. Par exemple, voici un programme qui efface toutes les occurrences de "zap me" :

```
(* rules section *)
rule token = parse
  | "zap␣me"    { token lexbuf }        (* ignore this token: no
    processing and continue *)
  | _ as c      { print_char c; token lexbuf }
```

FIG. 1.6 – Suppression de "zap me"

Exemple d'un programme qui compresse les espaces et tabulations multiples en un seul espace, et qui supprime un espace en fin de ligne :

```
{}
rule token = parse
  | [' ' '\t']+          { print_char ' '; token lexbuf }
  | [' ' '\t']+ '\n'     { token lexbuf }      (* ignore this token *)
```

FIG. 1.7 – Compression

Les actions peuvent inclure du code Ocaml qui renvoie une valeur. Chaque fois que la fonction d'analyseur lexical est appelée, elle continue de s'appliquer sur les *tokens* depuis lesquels il a analysé en dernier jusqu'à atteindre la fin du fichier.

Remarque : ce que fait l'analyseur, c'est simplement de vérifier que l'input vérifie les règles lexicales, il n'y a pas de "résultat" à proprement parlé, mais simplement, en cas de succès, l'assurance que l'analyseur reconnaît l'input. Les actions permettent d'agir sur les mots reconnus au cours de l'analyse.

Les actions sont évaluées après que le *lexbuf* soit associé au buffer lexical courant et à l'identifiant qui suit le mot clé (ou à la chaîne de caractères correspondante). L'utilisation du lexbuf est fournie par la librairie standard du module Lexing dont la table 1.2 présente quelques extraits.

Lexing.lexeme lexbuf	Valeur de la chaîne reconnue
Lexing.lexeme_char lexbuf n	nième caractère de la chaîne reconnue (début à 0)
Lexing.lexeme_start lexbuf	Position de la chaîne depuis le début de l'input (début à 0)
Lexing.lexeme_end lexbuf	Position de la chaîne depuis la fin de l'input
Lexing.lexeme_start_p lexbuf	Position de type position (voir doc version Ocaml 3.08)
entrypoint [exp1 ... expn] lexbuf	Appelle un autre lexer sur le point d'entrée donné. lexbuf doit être le dernier argument

TAB. 1.2 – Quelques fonctions du module Lexing

1.6 Output : le scanner généré

La sortie *output* générée est un fichier `".ml"` du même nom que celui invoqué par Ocamllex . Le fichier généré contient les fonctions de scanner, des tables utilisées par le scanner pour faire correspondre les tokens et des petites fonctions auxiliaires. Les fonctions de scanner sont déclarées de la façon suivante (1.8) :

```
let entrypoint [arg1 ... argn] lexbuf =
    ...
and ...
```

FIG. 1.8 – Le(s) point(s) d'entrée

où la fonction a n + 1 arguments. Les n arguments proviennent de la définition des règles. La fonction scanner résultante nécessite un argument supplémentaire, appelé lexbuf de type Lexing.lexbuf qui doit être le dernier argument.

Lorsqu'un point d'entrée est appelé, il scanne les tokens à partir de l'argument lexbuf. Quand il trouve un motif correspondant, il exécute les actions correspondantes et retourne le lexbuf (amputé du mot reconnu). De façon à continuer l'analyse lexicale après l'évaluation d'une action, il faut appeler la fonction de scanner récursivement.

1.7 Démarrage

Il est possible d'activer les règles de façon conditionnelle. Quand on veut activer l'autre règle, il suffit d'appeler le point d'entrée de l'autre fonction. Par exemple, l'input suivant est constitué de deux règles, l'une vérifiant les tokens et l'autre "sautant" les commentaires (1.9).

```
{}
rule token = parse
  | [' ' '\t' '\n']+
        (* skip spaces *)
        { token lexbuf }
  | "(*"
        (* activate "comment" rule *)
        { comment lexbuf }
  ...
and comment = parse
  | "*)"
        (* go to the "token" rule *)
        { token lexbuf }
  | _
        (* skip comments *)
        { comment lexbuf }
  ...
```

FIG. 1.9 – Règles conditionnelles

Lorsque le scanner généré rencontre un commentaire ouvrant "(*" à la règle token, il active l'autre règle de commentaire. Lorsqu'il rencontre un commentaire fermant "*)" dans la règle comment, il retourne à la règle token.

1.8 Interface avec Ocamlyacc

L'un des principaux usages de Ocamllex est l'association avec son compagnon Ocamlyacc parseur-générateur. Les parseurs d'Ocamlyacc appellent l'une des fonctions de scanner pour trouver le token d'entrée suivant. La fonction est censée renvoyer le type du token suivant avec sa valeur associée. Pour utiliser Ocamllex avec Ocamlyacc , les fonctions de scanner doit utiliser un module de parseur pour définir les types de token, qui sont défini dans les attributs '%token' apparaissant dans l'input de Ocamlyacc . Par exemple, si le fichier d'entrée de Ocamlyacc est parse.mly et que l'un des token est "NUMBER",une partie du scanner devrait ressembler à :

```
{
  open Parse
}

rule token = parse
  ...
  | ['0'-'9']+ as num { NUMBER (int_of_string num) }
  ...
```

FIG. 1.10 – Utilisation token lex - yacc

1.9 Options

Voici quelques options pour Ocamllex (table 1.3) :

−o output−file	Change le nom de la sortie
−ml	L'automate est codé avec des fonctions ocaml plutôt qu'un automate built-in. Pour le débuggage.
−q	Pour supprimer les messages d'information.

Tab. 1.3 – Options d'Ocamllex

Chapitre 2

Ocamlyacc

2.1 Introduction

Ocamlyacc est un générateur de parseur qui converti une grammaire (type LALR(1)) en un programme Ocaml qui parse cette grammaire. Quand on maîtrise Ocamlyacc , on peut l'utiliser pour construire des parsers d'une large gamme de langages, de la simple calculatrice au langage de programmation complexe.

Ocamlyacc est très proche de yacc (bison) qui sont répandus dans les environnements de programmation C. La maîtrise d'ocaml est nécessaire pour l'utiliser.

Ce tutoriel est fait de chapitres simples qui expliquent les concepts de base et présentent quelques exemples.

2.2 Concepts

2.2.1 Langages et grammaires hors-contexte (type 2)

Pour que Ocamlyacc parse un langage, le langage doit être décrit par une grammaire hors-contexte (type 2). Ce qui veut dire qu'on doit spécifier un ou plusieurs groupes syntaxiques et donner les règles de constructions correspondantes. Par exemple, le langage C, une sorte de groupe syntaxique est appelé "expression". Un règle pour construire une expression peut être "un expression peut être construite à l'aide du signe - et d'une autre expression". Une autre peut être "une expression peut être un entier". Comme on peut le voir dans ces exemples, les règles sont souvent récursives, mais il doit y avoir au moins une règle qui mène hors de la récursion.

Le système formel le plus commun pour présenter les règles de façon lisible est la norme Backus-Naur Form ou "BNF". Toute grammaire exprimée en BNF est une grammaire hors-contexte. L'entrée d'Ocamlyacc est essentiellement BNF.

Seules les grammaires LALR(1) peuvent être traitées par Ocamlyacc . Pour simplifier, on doit pouvoir parser n'importe quel mot de l'entrée avec un seul token d'avance, ce qui est une description d'une grammaire LR(1), les grammaires LALR(1) rajoutent des restrictions plus complexes à expliquer, mais il est rare qu'une grammaire LR(1) ne soit pas LALR(1). Pour plus d'information à ce propos, il faut se référer à la sous-section 2.5.7.

Dans une règle de grammaire formelle pour un langage, chaque unité ou groupe syntaxique est désigné par un symbole. Les symboles non-terminaux désignent ceux qui sont construits par des unités ou groupes plus petits, les symboles terminaux (ou token) ne peuvent pas être subdivisés. Un morceau/mot de l'entrée qui correspond à un symbole terminal est un token, tandis que s'il correspond à un non-terminal, il se dénomme un groupe/groupement.

En utilisant le langage C comme exemple, voici ce que symboles terminaux et non-terminaux veulent dire. Les token de C sont les identifiants, les constantes (numérique et chaîne de caractères), les différents mots clés, les opérateurs arithmétiques et les ponctuations. La grammaire de C inclue les symboles terminaux suivants : "identifier", "number", "string", un symbole par mot clé, opérateur ou ponctuation ("if", "return", "const", "static", "int", "char", "plus-sign", "open-brace", "comma"...).

La figure 2.1 présente une simple fonction C subdivisée en tokens :

```
int             /* mot clé "int" */
square (int x)  /* identifier, open-paren, identifier, identifier,
   close-paren */
{               /* open-brace */
  return x * x; /* mot clé "return", identifier, asterisk, identifier
     , semicolon */
}               /* close-brace */
```

FIG. 2.1 – Simple fonction C traduite en tokens

Les groupes syntaxiques de C incluent expression, statement, declaration et définition de fonction. Ils peuvent être représentés dans la grammaire de C par les symboles non-terminaux "expression", "statement", "declaration" et "function definition". La grammaire complète utilisent de nombreux constructeurs supplémentaire, chacun ayant son symbole non-terminal de façon à exprimer ces quatres groupes syntaxiques. L'exemple précédent (figure 2.1) est une "function definition", composée d'une "declaration" et d'un "statement". Dans ce "statement", chaque x est une "expression" ainsi que $x * x$.

Chaque symbole non-terminal doit avoir une règle grammatical montrant comment il est construit à partir de plusieurs règles plus simples. Par exemple, un statement en C est le return ; il serait décrit avec une règle de grammaire qui pourrait s'écrire "un 'statement' peut être construit d'un 'return', une 'expression' et un point virgule ('semicolon')". Et ainsi pour tous les statements possibles en C.

Il faut distinguer un symbole non-terminal : celui qui énonce le langage complet. Il s'appelle le symbole de départ (start symbol). Dans un compilateur, cela signifie l'entrée complète du programme. En C, le symbole non-terminal "séquence de définitions et déclarations" joue ce rôle.

Par exemple, $1 + 2$ est une expression C valide (un partie valide d'un programme C), mais n'est pas valide en tant que programme C entier. Dans la grammaire hors-contexte de C, $1 + 2$ qui est une expression n'est pas le symbole de départ.

Le parser Ocamlyacc , lit en séquence les tokens comme les entrées et regroupe les tokens en utilisant les règles de grammaires. Si l'entrée est valide, la totalité des tokens est réduite à un seul groupe dont le symbole est le symbole de départ de la grammaire. Si l'entrée est invalide, le parseur renvoie une erreur. En C, l'entrée doit être une "séquence de définitions et déclarations".

2.2.2 Des règles formelles à l'entrée Ocamlyacc

Une grammaire formelle est une construction mathématique. Pour définir le langage pour Ocamlyacc , il faut créer un fichier exprimant la grammaire dans la syntaxe Ocamlyacc : un fichier de grammaire Ocamlyacc (voir la section sur le format des fichiers Ocamlyacc 2.3)

Un symbole non-terminal d'une grammaire formelle est représenté dans l'input Ocamlyacc par un identifier, comme en Ocaml. C'est un symbole Caml, excepté qu'il ne peut pas finir par

' et qu'il doit commencer par une minuscule (ex : expr, stmt, declaration).

La représentation Ocamlyacc d'un symbole terminal est aussi appelé type de token. Les types de token doivent être déclaré dans la section déclaration d'Ocamlyacc et ils doivent être ajoutés comme constructeurs pour des types concrets de token. En tant que constructeurs, ils doivent commencer par une majuscule (ex : Integer Identifier , IF ou RETURN). Le symbole terminal d'erreur est réservé pour la récupération d'erreurs (voir la section sur les symboles 2.3.2).

Les règles de grammaire ont également une expression en syntaxe Ocamlyacc . Dans l'exemple suivant, une règle Ocamlyacc possible pour le return de C :
stmt: RETURN expr SEMICOLON ;

Voir la section sur la syntaxe de règles de grammaire 2.3.3.

2.2.3 Valeurs sémantiques

La grammaire formelle sélection les tokens seulement d'après leur classification. Par exemple, sur une règle mentionne le symbole terminal 'integer constant', cela signifie que n'importe quel entier constant est valide pour cette position. La valeur précise de la constante est ignorée lors du parsing de l'entrée : si $x+4$ est grammaticalement correct, alors $x+1$ ou $x+3989$ est également grammaticalement correct.

Par contre, la valeur précise est très importante pour ce que l'entrée signifie une fois parsée. Un compilateur est inutile s'il échoue à distinguer entre les constantes 4, 1 et 3989 dans le programme. C'est pourquoi chaque token de la grammaire Ocamlyacc possède un type de token et une valeur sémantique. Voir la section sur la sémantique 2.3.5.

Un type de token est un symbole terminal définit dans la grammaire, tel que INTEGER , IDENTIFIER ou SEMICOLON . Il contient les informations nécessaire pour décider où la validité du token peut apparaître et comment le regrouper avec les autres tokens. Les règles de grammaire ne considère que le type des tokens, pas les tokens eux-mêmes.

La valeur sémantique contient le reste des l'information à propos des tokens, par exemple la valeur d'un entier, le nom d'un identifier. Attention, un token comme SEMICOLON n'a pas de valeur sémantique.

Par exemple, un token d'entrée peut être classé comme un token de type INTEGER et avoir une valeur sémantique 4. Un autre token d'entrée peut avec le même type de token INTEGER mais la valeur 3989. Lorsque une règle de grammaire indique qu'un INTEGER est autorisé, ces deux tokens sont acceptable parce qu'ils sont tous deux des INTEGER . Lorsque le parser accepte le token, il garde une trace de la valeur sémantique.

Chaque groupe peut aussi avec une valeur sémantique comme son symbole non-terminal. Par exemple, dans une calculatrice, une expression a typiquement une valeur sémantique qui est un nombre. Dans un compilateur, pour un langage de programmation, une expression a typiquement une valeur sémantique qui est une structure d'arbre décrivant le sens de l'expression.

2.2.4 Actions sémantiques

Pour qu'un programme soit utile, il doit être plus que correct à parser. Il doit aussi produire des sorties basées sur les entrées... Dans une grammaire Ocamlyacc , une règle de grammaire peut avoir une action associée écrite en Ocaml. Chaque fois que le parseur reconnaît une correspondance pour une règle, une action est exécutée. Voir la sous-section sur les actions 2.3.5.

La plupart du temps, l'objectif de l'action est de calculer la valeur sémantique globale de la construction globale à partir de ses parties. Par exemple, si on a une règle qui dit qu'une expression peut être la somme de deux expression, lorsque le parseur reconnaît une telle somme,

chaque sous-expression possède une valeur sémantique qui décrit comment elle a été construite. L'action d'une telle règle devrait créer une valeur du même genre pour l'expression reconnue.

Par exemple, voici figure 2.2 une règle qui dit qu'une expression peut être la somme de deux expressions :

```
expr: expr PLUS expr    { $1 + $3 }
        ;
```

FIG. 2.2 – Règle de somme

L'action indique comment produire la valeur sémantique de la somme des expressions à partir des valeurs sémantiques des deux sous-expressions.

2.2.5 Positions

De nombreuses applications, tels interpréteurs ou compilateurs, doivent produire des messages d'erreurs utiles et détaillés. Afin de faire cela, on doit être capable d'identifier la position dans un texte de chaque constructeur syntaxique. Ocamlyacc permet de faire cela.

Chaque token a une valeur sémantique et une position associée. Mais le type de position est le même pour tous les tokens et groupes. De plus, la sortie du parser est faite de structure de données permettant de stocker les positions. Voir la sous-section position 2.3.6.

Many applications, like interpreters or compilers, have to produce verbose and useful error messages. To achieve this, one must be able to keep track of the textual position, or location, of each syntactic construct. Ocamlyacc provides a mechanism for handling these locations. On peut utiliser ces positions pour atteindre les actions utilisant les fonctions du module Parsing.

2.2.6 Sortie d'Ocamlyacc : le fichier du parser

Lorsqu'on exécute Ocamlyacc , on donne un fichier de grammaire Ocamlyacc comme entrée. La sortie est un fichier Ocaml qui parse le langage décrit par la grammaire. Ce fichier est appelé un parseur Ocamlyacc . Attention Ocamlyacc est l'outil dont la sortie est le parseur Ocamlyacc

La fonction du parseur Ocamlyacc est de regrouper les tokens dans les groupes selon les règles de grammaire. Par exemple, de construire identifier et opérateurs en expression. Lorsqu'il fait ce regroupement, il effectue les actions des règles de grammaires qu'il utilise.

Les tokens viennent d'une fonction appelée l'analyseur lexical que l'on doit fournir d'une façon ou d'une autre. Le parseur Ocamlyacc appelle l'analyseur lexical chaque fois qu'il a besoin d'un nouveau token. Il ne sait pas ce que contient les tokens, (bien que leur valeur sémantique peut être récupérée). Typiquement, l'analyseur lexical fabriquent les tokens en parsant les caractères d'un texte, tandis qu'Ocamlyacc ne dépend pas de ça. Voir la section 2.4.2 sur la fonction d'analyse lexicale.

Le fichier parseur de Ocamlyacc est en Ocaml. Il définit les fonctions qui implémentent la grammaire. Chaque fonction d'entrée du code Ocaml généré est nommé d'après les symboles de départ du fichier de grammaire. Les fonctions ne constituent pas un programme Ocaml complet : il faut fournir quelques fonctions supplémentaires. La première est l'analyseur lexical qui doit être donné comme argument de la fonction d'entrée du parseur. Une autre fonction est la fonction de rapport d'erreur appelée par le parseur afin de rapporter une erreur. Et enfin, un programme complet Ocaml doit pouvoir appeler un (ou plusieurs) fonction d'entrée générée ou alors le parseur ne pourra pas d'exécuter. Voir la section 2.4 sur l'interface du parseur.

2.2.7 Les étapes pour utiliser Ocamlyacc

Voici les étapes de construction d'un langage utilisant Ocamlyacc , partant de la spécification de la grammaire allant jusqu'au compilateur ou interpréteur :

- Formellement spécifier la grammaire sous une forme reconnue par Ocamlyacc (voir la section sur les fichiers de grammaire 2.3). Pour chaque règle dans le langage, décrire l'action qui doit être effectuée lorsqu'une instance de cette règle sera reconnue. L'action est décrite par une séquence de statement Ocaml.
- Ecrire un analyseur lexical pour analyser l'entrée et transmettre les tokens au parseur. L'analyseur lexical peut être écrit à la main en Ocaml (voir la section 2.4.2). Il peut aussi être produit par Ocamllex , voir la partie 1.
- Ecrire une fonction de contrôle qui appelle le parseur produit par Ocamlyacc
- Ecrire les fonctions de rapport d'erreurs.

Pour transformer ce code en exécutable, il faut suivre les étapes suivantes :

- Executer Ocamlyacc sur la grammaire pour produire le parseur.
- Compiler le code de sortie d'Ocamlyacc ainsi que les autres fichiers sources.
- Lier les fichiers objets pour produire le produit fini.

2.3 Le fichier de grammaire Ocamlyacc

Ocamlyacc prend en entrée une spécification de grammaire hors-contexte et produit un fonction Ocaml qui reconnaît les instances correctes de cette grammaire. Le fichier d'entrée de grammaire Ocamlyacc a par convention une extension en ".mly". Voir la section Invoquer Ocamlyacc 2.8.

2.3.1 Structure du fichier

Le fichier de grammaire Ocamlyacc est constitué de quatre sections principales, présentées ici (figure 2.3) avec leurs délimiteurs :

```
%{
        Header - Ocaml declarations (Ocaml code)
%}
        Ocamlyacc declarations
%%
        Grammar rules
%%
        Trailer - Additional Ocaml code (Ocaml code)
```

FIG. 2.3 – Structure du fichier de grammaire Ocamlyacc

- Les `%%`, `%{` et `%}` sont des ponctuations qui permettent de séparer les sections.
- Dans la section *header* on définit les types, variables et fonctions utilisées dans les actions.
- Dans la section *Ocamlyacc declarations*, on déclare les noms des symboles terminaux et non-terminaux. On peut aussi décrire les précédences des opérateurs et les types des données des valeurs sémantiques des différents symboles.
- La section *grammar rules* définit comment construire chaque non-terminal à partir de ses morceaux
- La section *Trailer* peut contenir du code Ocaml

Par défaut, les commentaires sont encadrés entre "/*" et "*/" (comme en C) excepté dans le code Ocaml. On utilise donc "/*" et "*/" dans les déclarations et dans les sections de règles, et "(*" et "*)" dans les sections de header et de trailer.

Le Header

On met dans le header les déclarations de fonctions ou variables qui sont utilisées dans les actions des règles de grammaire. Elle sont copiées au début du fichier de parser afin qu'elle précèdent le définition de la fonction de parser. On peut ouvrir un autre module dans cette section. Si on n'utilise pas de déclaration Ocaml, on peut omettre le %{ et %} qui encadrent cette section.

Les déclarations

La section de déclarations Ocamlyacc contient les déclarations qui définissent les symboles terminaux et non-terminaux, spécifient les précédences et ce genre de choses. Il doit y avoir au minimum un **%start** et les directives **%type** correspondantes. Voir la sous-section 2.3.7 sur les déclarations.

Les règles

La section des règles de grammaire contient un ou plusieurs règles de grammaire Ocamlyacc et rien d'autre. Voir la sous-section 2.3.3 sur la syntaxe des règles de grammaire.

Il doit toujours y avoir au moins une règle de grammaire, et le premier %% (celui qui précède les règles de grammaire) ne peut pas être omis, même si c'est la première ligne du fichier.

Le Trailer

La section trailer sera copiée telle quelle à la fin du fichier de parser, (de la même façon que le header est copié tel quel en début de fichier du parser). C'est l'endroit où mettre le code Ocaml qui n'est pas nécessaire de mettre avant les règles. Voir la section 2.4 de l'interface du parser.

Si cette dernière section est vide, on peut omettre le %% qui le sépare des règles de grammaire.

2.3.2 Symboles, terminaux et non-terminaux

Les *symboles* de grammaires Ocamlyacc représentent les classifications grammaticales des langages.

Un *symbole terminal* (appelé type de token) représente une classe de tokens syntaxiquement équivalents. On utilise le symbole dans les règles de grammaire pour signifier qu'un token est alloué dans cette classe. Le symbole est représenté dans le parseur Ocamlyacc par une valeur de type variant, et la fonction lexer renvoie une type de token pour indiquer quelle sorte de token a été lu.

Un *symbole non-terminal* représente une classe de groupe syntaxique équivalents. Le nom du symbole est utilisé pour écrire les règles de grammaire. Il doit commencer par une minuscule.

Les noms de symbole contiennent des lettres, des chiffres (sauf en début de mot), des sous-lignés.

Le symbole terminal de la grammaire est un type de token qui est une valeur de type variant en Ocaml. Il doit donc commencer par une minuscule. Chaque nom doit être défini dans la section déclaration avec **%token**. Voir la section (2.3.7) sur les noms des types de token.

La valeur renvoyée par la fonction de lexer est toujours l'un des symboles terminaux. Chaque type de token devient une valeur de type variant dans le fichier parser, afin que la fonction de lexer puisse en retourner un.

Parce que la fonction de lexer est définie dans un fichier séparé, il faut faire en sorte que les définitions de types de token soient disponibles à ce niveau. Après avoir invoqué `ocamlyacc filename.mly`, le fichier "filename.mli" qui est généré contient les définitions de types de token. Il est utilisé dans la fonction de lexer.

Le symbole d'erreur est un symbole terminal réservé pour récupérer les erreurs (voir la section 2.6 sur le rattrapage d'erreur), il doit rester réserver à cet usage.

2.3.3 Syntaxe des règles de grammaire

Un règle de grammaire Ocamlyacc a la forme générale suivante (figure 2.4) :

```
result:
          symbol ... symbol { semantic-action }
        | ...
        | symbol ... symbol { semantic-action }
        ;
```

FIG. 2.4 – Structure de règle de grammaire Ocamlyacc

où `result` est le symbole non-terminal que cette règle décrit, `symbol` sont des symboles terminaux ou non-terminaux qui sont assemblés par cette règle (voir la section 2.3.2 sur les symboles).

Par exemple la règle suivante (figure 2.5) que deux groupes de type `exp` avec un token `PLUS` au milieu peuvent être combiné en un groupe plus large de type `exp`.

```
exp:      exp PLUS exp          {}
        ;
```

FIG. 2.5 – Exemple de la combinaison par PLUS

Les espaces dans les règles sont nécessaires seulement pour séparer les symboles, des espaces supplémentaires peuvent être ajoutés.

A la suite des composants de la règle, il doit y avoir l'action qui détermine la sémantique de la règle. Un action doit être définie de cette façon `{ Ocaml code}`, voir la section 2.3.5 sur les actions.

Il est possible d'écrire des règles séparées pour le même résultat, mais on peut aussi les rejoindre à l'aide d'une barre verticale "|" comme dans la figure 2.6. Les règles sont considérées

```
result:
          rule1-symbol ... rule1-symbol { rule1-semantic-action }
        | rule2-symbol ... rule2-symbol { rule2-semantic-action }
        | ...
        ;
```

FIG. 2.6 – Plusieurs règles

comme distinctes, même quand elles sont écrites de cette manière.

Si la composition d'une règle est vide, cela veut dire que le résultat peut correspondre à une chaîne de caractère vide. Par exemple, voici comment définir une séquence de zéro séparée de virgules ou de plus de groupe d'exp (figure 2.7) :

```
expseq:    /* empty */    {}
         | expseq1                {}
         ;

expseq1:   exp                      {}
         | expseq1 COMMA exp                {}
         ;
```

FIG. 2.7 – Règle vide

L'usage veut qu'on écrive le commentaire `/* empty */` pour chaque règle sans composant.

2.3.4 Règles récursives

Une règle est appelée récursive si son résultat non-terminal apparaît aussi dans le membre de droite de la règle. Pratiquement toutes les grammaires Ocamlyacc utilisent la récursion, parce que c'est le seul moyen de définir une séquence d'un nombre indéterminé d'une chaîne de caractère. Voici l'exemple (figure 2.8) de la définition récursive d'une séquence séparée de virgule d'une ou plusieurs expressions :

```
expseq1:   exp                      {}
         | expseq1 COMMA exp                {}
         ;
```

FIG. 2.8 – Règle récursive - virgules

Cette récursion est appelée récursion gauche car l'utilisation du symbole `expseq1` est le symbole le plus à gauche dans le membre de droite de la règle. Voici la même construction avec une récursion droite (figure 2.9) :

```
expseq1:   exp                      {}
         | exp COMMA expseq1                {}
         ;
```

FIG. 2.9 – Règle récursive droite

Toute sorte de séquence peut être définie en utilisant la récursion gauche ou droite, mais il est préférable de toujours utiliser la récursion gauche car cela utilise un nombre limité d'espace dans la pile. La récursion droite utilise un espace dans la pile Ocamlyacc proportionnel au nombre d'élément de la séquence, parce que tous les éléments doivent être empilés avant que la règle puisse être appliquée au moins une fois. Voir la section 2.5 sur l'algorithme du parser.

On peut aussi trouver de la récursion mutuelle ou indirecte lorsque le résultat de la règle n'apparaît pas directement dans le membre droit, mais qu'il apparaît dans des règles d'autres non-terminaux qui apparaissent eux dans le membre droit de cette règle.

Par exemple dans la figure 2.10 suivante, on définit deux non-terminaux mutuellement récursifs, chacun faisant référence à l'autre.

```
expr:     primary                          {}
        | primary PLUS primary   {}
        ;

primary:  constant                         {}
        | LPAREN expr RPAREN     {}
        ;
```

FIG. 2.10 – Règles mutuellement récursives

2.3.5 Définir la sémantique d'un langage

Les règles de grammaire d'un langage ne déterminent que la syntaxe. La sémantique est déterminée par les valeurs sémantiques associées aux différents tokens et groupes, et par les actions à effectuer lorsque les différents groupes sont reconnus.

Par exemple, un calculateur calcule correctement parce que la valeur associée à chaque expression est le nombre correct. Il ajoute correctement parce que l'action du groupe $x + y$ est d'ajouter les nombres associés à x et y.

Type de données des valeurs sémantiques

Dans un simple programme, il peut être suffisant d'utiliser le même type de données pour les valeurs sémantiques de tous les constructeurs du langage. Dans la plupart des programmes, il est cependant nécessaire d'avoir des types de données différents pour des types de tokens ou de groupe différents. Par exemple, une constante numérique peut nécessiter le type int ou float alors qu'une constante chaîne de caractère ou un identifiant peut nécessiter le type string.

Afin d'utiliser plus d'un type de données pour les valeurs sémantiques dans un parser, Ocamlyacc impose de choisir un type de données pour chaque symbole (terminal ou non-terminal) pour lequel la valeur sémantique est utilisée. Pour les tokens, cela se fait à l'aide de `%token` Ocamlyacc declaration (voir la sous-sous-section 2.3.7 sur les noms de type de token) et pour les groupes avec `%type` Ocamlyacc declaration (voir la sous-sous-section 2.3.7 sur les symboles non-terminaux).

Actions

Une action est associée à une règle syntaxique est contient du code Ocaml qui est exécuté chaque fois qu'une instance de la règle est reconnu. La tâche de la plupart des actions est de calculer une valeur sémantique pour le groupe construit par la règle à partir des valeurs sémantiques des tokens et les plus petits groupes qui constituent la règle.

Une action est construite de statement Ocaml encadré par des accolades. Les règles n'ont qu'une action à la fin de la règle suivie par tous les composants.

Le code Ocaml dans une action peut faire référence aux valeurs sémantiques des composants de la règle en les faisant correspondre avec `$n`, valeur du nième composant. La valeur de cette évaluation de l'action est la valeur du groupe construit.

L'exemple classique (figure 2.11) :

```
exp:      ...
        | exp PLUS exp { $1 +. $3 }
```

FIG. 2.11 – Action

Cette règle construit une `exp` à partir de deux groupes `exp` plus petits connectés avec le token du signe "+". Dans l'action `$1` et `$3` font référence aux valeurs sémantiques des deux composants des groupes `exp`, qui sont le premier et le troisième symboles du membre droit de la règle. La somme est renvoyée afin que la valeur sémantique des l'expression d'addition soit reconnue par la règle. S'il y avait une valeur sémantique utilisable avec le token PLUS, on pourrait y référer avec `$2`.

2.3.6 Suivre les positions

Bien que les règles de grammaire et les actions sémantiques soient suffisantes pour écrire un parseur entièrement fonctionnel, il peut être utile de pouvoir rajouter des informations complémentaires, en particulier les symboles de positions.

La façon dont les positions sont traitées est définie en fournissant le type de données et les actions à effectuer quand les règles sont appliquées.

Type de données des positions

Le contenu de cette section est valable depuis Ocaml 3.08.

Le type de données pour les positions ce construit comme ceci (figure 2.12) :

```
type position = {
    pos_fname : string;         (* file name *)
    pos_lnum : int;             (* line number *)
    pos_bol : int;              (* the offset of the beginning of the
        line *)
    pos_cnum : int;             (* the offset of the position *)
}
```

FIG. 2.12 – Type de positions

La valeur du champ `pos_bol` est le nombre de caractères entre le début du fichier et le début de la ligne, la valeur du champ `pos_cnum` est le nombre de caractères entre le début du fichier et la position.

Le moteur lexical parvient seulement au champ de `pos_cnum` du `lexbuf.lex_curr_p` avec le nombre de caractères lus à partir du début du lexbuf. Le programmeur est donc responsable de l'exactitude des autres champs. Avant d'utiliser la position dans le parseur, il faut initialiser `Lexing.lexbuf.lex_curr_p` correctement dans le lexer, en utilisant une fonction comme celle ci (figure 2.13) :

Actions et positions

Les actions sont utiles non seulement pour définir la sémantique du langage mais aussi (à l'aide des positions) pour décrire le comportement du parseur. Le moyen le plus simple de construire des positions de groupes syntaxique est très similaire à la façon dont les valeurs

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

FIG. 2.13 – Initialiser les positions du lexer/parser

sémantiques sont calculées. Pour une règle donnée, plusieurs constructeurs peuvent être utilisés pour accéder aux positions des éléments correspondants. La position du nième composant du membre droit peut être obtenu à l'aide de (figure 2.14) :

```
val Parsing.rhs_start : int -> int
val Parsing.rhs_end : int -> int
```

FIG. 2.14 – Position

`Parsing.rhs_start n` renvoie l'écart avec le premier caractère du nième item du membre droit de la règle. `Parsing.rhs_end n` renvoie l'écart avec le dernier caractère de l'item. Ces fonctions doit être appelées par les actions. n vaut 1 pour l'item le plus à gauche et les premier caractère dans un fichier a un écart de 0.

On peut aussi utiliser les fonctions suivantes (figure 2.15) :

```
val Parsing.rhs_start_pos : int -> Lexing.position
val Parsing.rhs_end_pos : int -> Lexing.position
```

FIG. 2.15 – Fonctions de position

(Depuis Ocaml 3.08) Les fonctions suivantes retournent une position plutôt qu'un écart (voir sous-sous-section 2.3.6).

La position du groupe du membre de gauche peut être obtenue par (figure 2.16) :

```
val Parsing.symbol_start : unit -> int
val Parsing.symbol_end : unit -> int
```

FIG. 2.16 – Fonctions pour membre de gauche

`symbol_start ()` renvoie l'écart avec le premier caractère du membre gauche et `symbol_end ()` l'écart avec le dernier caractère.

(Depuis Ocaml 3.08) Les fonctions suivantes sont comme `symbol_start` and `symbol_end`, excepté qu'elles renvoient une position plutôt qu'un écart (voir sous-sous-section 2.3.6).

```
val Parsing.symbol_start_pos : unit -> Lexing.position
val Parsing.symbol_end_pos : unit -> Lexing.position
```

Voici un exemple basique qui utilise le type de donnée par défaut des positions (figure 2.17) :

```
exp:      ...
        | exp DIVIDE exp
                { if $3 <> 0.0 then $1 /. $3
                  else (
                    let start_pos = Parsing.rhs_start_pos 3 in
                    let end_pos = Parsing.rhs_end_pos 3 in
                    printf "%d.%d-%d.%d: division by zero"
                      start_pos.pos_lnum (start_pos.pos_cnum -
                          start_pos.pos_bol)
                      end_pos.pos_lnum (end_pos.pos_cnum - end_pos.
                          pos_bol);
                    1.0
                  )
                }$
```

FIG. 2.17 – Exemple d'utilisation des positions

2.3.7 Déclarations Ocamlyacc

La section "declaration Ocamlyacc " de la grammaire Ocamlyacc définit les symboles utilisés pour formuler la grammaire et les types de données des valeurs sémantiques. Voir la sous-section 2.3.2 sur les symboles.

Tous les types de tokens doivent être déclarés. Les symboles non-terminaux doivent être déclarés s'il est nécessaire de spécifier quel type de données utiliser pour la valeur sémantique (voir la sous-sous-section 2.3.5 sur les types de données des valeurs sémantiques).

La première règle dans le fichier spécifie aussi le symbole de départ, par défaut. Si on veut un autre symbole comme départ, on doit le déclarer explicitement (voir la sous-section 2.2.1 sur les grammaires hors-contexte).

Noms des types de token

Le moyen classique pour déclarer un nom de type de token (symbole terminal) est le suivant (figure 2.18) :

```
%token name ... name
%token <type> name ... name
```

FIG. 2.18 – Déclaration de type de token

Ocamlyacc converti cela sous forme d'un type dans le parseur, de façon à ce que la fonction lexer puisse utiliser le nom correspondant au code du type de ce token.

Si le token a une valeur, l'argument de la déclaration **%token** doit inclure le type de données encadré par les "<>" (voir la sous-sous-section 2.3.5 sur les types de données des valeurs sémantiques).

Par exemple : **%token** <float> NUM /* define **token** NUM **and** its **type** */

La partie "type" doit être une expression de type Caml. La partie <type> est copiée dans le fichier sortie en ".mli", tous les noms de constructeur de type doivent être valides (par exemple. Module_name.type_name).

Opérateurs de précédences

On utilise les déclarations %left, %right ou %nonassoc pour spécifier les précédences et associativités. Voir la section 2.5.3 sur les conflits résolus par les opérateurs de précédence.

La syntaxe de déclaration de précédence est la suivante (figure 2.19) :

```
%right symbols ... symbols
%nonassoc symbols ... symbols
```

Fig. 2.19 – Syntaxe des déclarations de précédence

Les déclarations de précédence précisent l'associativité et la précédence relative pour tous les symboles :

- L'associativité d'un opérateur "op" détermine comment interpréter la répétition de l'opérateur : si "x op y op z" est considéré comme "(x op y) op z" ou "x op (y op z)". %left spécifie l'associativité gauche "(x op y) op z)" et %right spécifie l'associativité droite "x op (y op z)". %noassoc ne spécifie pas d'association : "x op y op z" sera considéré comme une erreur de syntaxe.
- La précédence d'un opérateur détermine son comportement lorsqu'il est encapsulé avec d'autres opérateurs. Tous les tokens déclarés dans la même déclaration de précédence ont une précédence équivalente, ce qui signifie qu'il seront encapsulé selon leur associativité. Lorsque deux tokens déclarés dans deux déclarations de précédence différente sont associés, celui déclaré le plus tard possède la précédence la plus forte, il est donc groupé (au sens syntaxique) en premier.

Symboles non-terminaux

On peut déclarer le type de chaque symbole non-terminal pour lesquels les valeurs sont utilisées. Cela se fait à l'aide de la déclaration %type comme ça :

%type <type> nonterminal ... nonterminal. "nonterminal" est le nom d'un symbole non-terminal et "type" est le nom du type voulu. On peut donner autant de symboles non-terminaux de départ que l'on souhaite dans la même déclaration %type s'ils ont le même type. On utilise les espaces pour séparer les noms de symboles.

Cette déclaration est nécessaire pour les symboles de départ. Voir la sous-sous-section 2.3.7 sur les noms de types de token.

Le symbole de départ

On doit déclarer un symbole de départ/démarrage en utilisant la déclaration %start de la façon suivante :

%start symbol ... symbol

Chaque symbole de départ a une fonction de parsing du même nom dans le fichier de sortie. On s'en sert comme point d'entrée pour la grammaire. Rappel : chaque symbole de départ doit avoir un type associé, on utilise alors la déclaration %type, voir sous-sous-section précédente 2.3.7 sur les symboles non-terminaux.

Récapitulatif des déclarations Ocamlyacc

Voici le récapitulatif des déclarations nécessaires pour définir une grammaire :
Here is a summary of the declarations used to define a grammar :

- **%token** Déclaration d'un symbole terminal (nom de type de token) sans précédence ni associativité (voir 2.3.7).
- **%right** Déclaration d'un symbole terminal (nom de type de token) qui a une associativité droite (voir 2.3.7).
- **%left** Déclaration d'un symbole terminal (nom de type de token) qui a une associativité gauche (voir 2.3.7).
- **%nonassoc** Déclaration d'un symbole terminal (nom de type de token) qui n'est pas associatif (erreur de syntaxe si c'était le cas, voir 2.3.7).
- **%type** Déclaration du type de la valeur sémantique d'un symbole non-terminal (voir 2.3.7).
- **%start** Précise le symbole de départ de la grammaire (voir 2.3.7).

2.4 L'interface du parser

Le parser Ocamlyacc est en fait un ensemble de fonction Ocaml nommées à partir des symboles de départ de la grammaire (voir sous-sous-section 2.3.7 sur les symboles de départ). Cette section décrit les conventions des fonctions du parser et les autres fonctions nécessaires.

2.4.1 La fonction Parser

Afin de faire un parsing, il faut appeler la fonction de parser avec deux paramètres. Le premier paramètre est la fonction d'analyseur lexical de type Lexing.lexbuf -> **token** et le second est une valeur de type Lexing.lexbuf.

Soit `parse` le symbole de départ dans le fichier `parser.mly` et `token` la fonction lexer dans le fichier `lexer.mll`, on procède de la manière suivante (figure 2.20) :

```
let lexbuf = Lexing.from_channel stdin in
...
let result = Parser.parse Lexer.token lexbuf in
...
```

FIG. 2.20 – Appel des fonctions

La fonction parser lit les tokens, exécute les actions et se termine soit quand elle rencontre un fin d'entrée, soit une erreur de syntaxe non récupérée.

2.4.2 La fonction d'analyse lexicale

La fonction d'analyse lexicale est nommée d'après les déclarations de règles. Elle reconnaît les tokens depuis le flux d'entrée et les renvoie au parser. Ocamlyacc ne crée pas cette fonction automatiquement : il faut l'écrire afin que la fonction de parser puisse l'appeler. On peut assimiler cette fonction à un scanner lexical. Elle est généralement générée par Ocamllex (voir partie 1 ou le chapitre 12 du manuel Ocaml `http ://caml.inria.fr/pub/docs/manual-ocaml/manual026.html`).

2.4.3 La fonction de rapport d'erreur

Le parser Ocamlyacc détecte les erreurs de parsing ou de syntaxe lorsqu'il lit un token qui ne satisfait aucune des règles de syntaxe. Un action de la grammaire peut également provoquer une erreur, utilisant la levée `Parsing.Parse_error`.

Le parser Ocamlyacc va rapporter l'erreur en appelant une fonction de rapport d'erreur appelée `parse_error` qui est optionnelle. Par défaut, la fonction `parse_error` ne fait aucune action. Elle est appelée par la fonction de parser lorsqu'une erreur de syntaxe est trouvée et reçoit un argument. Pour une erreur de parsing, l'erreur est généralement `syntaxe error`.

La définition suivante est généralement suffisante pour les programmes simples :

```
let parse_error s = print_endline s
```

Si la fonction `parse_error` renvoie quelque chose à la fonction de parsing, elle essayera de rattraper l'erreur s'il existe des règles (valides) de grammaire qui récupèrent les erreurs (voir la section 2.6 sur le rattrapage d'erreurs). Si la récupération est impossible, la fonction de parsing lèvera l'exception `Parsing.Parse_error`.

2.5 L'algorithme du parseur Ocamlyacc

Au fur et à mesure que Ocamlyacc lit les tokens, ils sont empilés avec leur valeur sémantique. La pile est appelée *pile du parser*. On appelle ça du *shifting*.

Par exemple, supposons que la calculatrice infixe a lu $1 + 5*$ et que 3 soit à lire. La pile aura 4 éléments, un par token shifté.

La pile n'a pas toujours un élément pour chaque token. Lorsque les derniers n tokens et groupes shiftés correspondent aux composants d'une règle de grammaire, ils peuvent être combinés suivant cette règle. C'est ce qu'on appelle la *réduction*. Ces tokens et groupes sont remplacés dans la pile par un simple groupe dont le symbole est le résultat (membre de gauche) de la règle. Exécuter l'action de la règle fait partie du processus de réduction, cela calcule la valeur sémantique du groupe résultant.

Par exemple, si la pile de la calculatrice infixe contient $1 + 5 * 3$ et que le prochain token est un saut de ligne, alors les trois (derniers) éléments peuvent être réduits à 15 en appliquant la règle `expr: expr MULTIPLY expr;`. La pile contiendra alors les trois éléments suivants : $1 + 15$. Un réduction supplémentaire peut être effectuée, qui produit la valeur 16. Et dès lors le saut de ligne peut être shifté.

Le parser essaie, en shiftant ou réduisant, de réduire l'input complètement à un groupe simple dont le symbole est le symbole de départ de la grammaire (voir la sous-section 2.2.1 sur les grammaires hors-contexte).

Ce genre de parser est connu dans la littérature comme un parser ascendant.

2.5.1 Les tokens en avant

Le parseur Ocamlyacc ne réduit pas toujours immédiatement, aussitôt que les n tokens et groupes correspondent à une règle. Une stratégie aussi simple de couvre pas la plupart des langages. En fait, lorsqu'une réduction est possible, le parseur regarde "en avant" le token suivant de façon à décider quoi faire.

Quand un token est lu, il n'est pas immédiatement shifté. Il devient d'abord un *token en avant*, qui n'est pas sur la pile. Le parseur peut alors effectué une ou plusieurs réductions de tokens et groupes sur la pile, tandis que le token en avant reste de côté. Quand plus aucune réduction de peut être effectuée, le token en avant est shifté sur la pile. Il peut donc rester des réductions possibles, et selon le type du token en avant, certaines règles peuvent choisir de retarder leur application.

Voici figure 2.21 un exemple de cas simple d'utilisation du token en avant. Ces deux règles définissent des expressions contenant l'opérateur d'addition et l'opérateur postfixe de factorielle (FACTORIAL pour "!") et l'encadrement par parenthèses.

```
expr:     term PLUS expr
        | term
        ;

term:     LPAREN expr RPAREN
        | term FACTORIAL
        | NUMBER
        ;
```

FIG. 2.21 – Token en avant

Supposons que les tokens 1 + 2 sont lus et shifté, que faire ensuite ?

- Si le token suivant est RPAREN alors les trois premiers tokens doivent être réduit comme expr. Ce sera alors la seule possibilité, parce que shifter RPAREN produira une séquence term RPAREN et aucune règle ne le permet.
- Si le token suivant est FACTORIAL, il doit être shifté directement afin de réduire 2 FACTORIAL en term. Si le parser voulait réduire avant de shifter 1 + 2 deviendrait une expr. On ne pourrait donc plus shifter FACTORIAL parce qu'on aurait sur la pile une séquence de symboles expr FACTORIAL, qui n'est pas autorisé dans les règles.

2.5.2 Conflits shift/reduce

Voici un exemple figure 2.22 de règles pour un langage ayant des if-then et if-then-else que l'on souhaite parser :

```
if_stmt:
          IF expr THEN stmt
        | IF expr THEN stmt ELSE stmt
        ;
```

FIG. 2.22 – Conflits shift/reduce

IF, THEN, et ELSE sont des symboles terminaux pour les mots clés correspondants. Quand le token ELSE est lu et devient un token en avant, le contenu de la pile (supposant l'entrée correcte) est valide pour une réduction par la première règle. Mais il est aussi légitime de shifter le ELSE parce que cela pourrait conduire à une éventuelle réduction par la deuxième règle.

Cette situation, lorsque shifter ou réduire peuvent être tout deux valides, est appelée un conflit shift/reduce. Ocamlyacc est conçu pour résoudre ce problème en choisissant de shifter, sauf s'il y a d'autres précisions dans les déclarations de précédence sur les opérateurs.

La raison pour laquelle on choisit plutôt de shifter que de réduire est exposée ci-après. Parce que le parser préfère shifter le ELSE le résultat est d'associer la clause "else" au statement "if" le plus imbriqué, en rendant les deux entrées suivantes équivalentes (figure 2.23) :

Il y a un conflit car la grammaire telle qu'elle est écrite est elle-même ambiguë : on peut parser un statement "if" simple imbriqué. Les conventions établies sont que les ambiguïtés sont

```
if x then if y then win (); else lose;

if x then do; if y then win (); else lose; end;
```

FIG. 2.23 – Entrées équivalentes si shift

résolues en associant les clauses "else" au statement "if" le plus imbriqué et c'est ce que fait Ocamlyacc en choisissant de shifter plutôt que de réduire.

Remarque : il serait idéalement plus propre d'écrire une grammaire non ambiguë, mais ce n'est pas aisé dans ce cas. Cette ambiguïté particulière a été rencontrée dans les spécifications d'*Algol 60* et est appelée l'ambiguïté "dandling else".

La définition précédente (figure 2.22) du `if_stmt` est la seule à blâmer pour le conflit, mais le conflit n'apparaît en fait qu'avec l'ajout d'autres règles. Voici un fichier figure 2.24 complet de grammaire Ocamlyacc qui provoque un conflit.

```
%token IF THEN ELSE variable
%%
stmt:     expr
        | if_stmt
        ;

if_stmt:
          IF expr THEN stmt
        | IF expr THEN stmt ELSE stmt
        ;

expr:     variable
        ;
```

FIG. 2.24 – Grammaire à conflit

2.5.3 Précédence des opérateurs

D'autres conflits shift/reduce apparaissent dans les expressions arithmétiques. Shifter n'est alors pas la meilleur solution. Les déclarations de précédence Ocamlyacc permettent de spécifier quand shifter ou quand réduire.

Cas de précédence nécessaire

Voici figure 2.25 une grammaire ambiguë (car l'entrée $1 - 2 * 3$ peut être parsée de deux façons) :

Si le parser voit les tokens $1 - 2$ devra-t-il réduire avec la règle de soustraction ? Cela dépend du token suivant.

- `)` on doit réduire, shifter n'est pas valide car aucune règle ne peut réduire la séquence `MINUS expr` (c'est-à-dire -2) ou n'importe quel groupe commençant par ça.
- $*$ ou $**$ on doit choisir : shifter ou réduire peuvent être effectuer mais avec des résultats différents.

```
expr:      expr MINUS expr
         | expr MULTIPLY expr
         | expr LT expr
         | LPAREN expr RPAREN
         ...
         ;
```

FIG. 2.25 – Grammaire à conflit soluble par précédence

Pour décider ce que Ocamlyacc doit faire, il faut regarder les résultats. Si l'opérateur suivant "op" est shifté, alors il doit être réduit d'abord pour permettre une autre possibilité de réduction pour le -. Le résultat serait 1 - (2 op 3). Sinon, si la soustraction est réduite avant de shifter l'opérateur, le résultat est (1 - 2) op 3. Ainsi, le choix entre shift et reduce doit dépendre de la relative précédence entre les opérateurs (ici "-" et "op").

Dans le cas où l'entrée est $1 - 2 - 5$, est-ce $(1 - 2) - 5$ ou $1 - (2 - 5)$ qui doit être reconnu ? Pour la plupart des opérateurs, on préfère la première possibilité, qu'on appelle associativité gauche. La deuxième, association droite, est préférée pour les opérateurs d'affectations. Le choix d'association gauche ou droite dépend si le parser choisit shift ou reduce lorsque la pile contient $1 - 2$ et que le token en avant est $-$.

Spécification des précédences d'opérateurs

Ocamlyacc permet de spécifier des choix à l'aide des déclarations de précédence **%left** et **%right** (voir la sous-sous-section 2.3.7 sur la précédence des opérateurs). Chaque déclaration contient une liste de token, qui sont des opérateurs dont on déclare la précédence et l'associativité. La déclaration **%left** rend les opérateurs associatifs gauche et **%right** associatifs droit. La troisième alternative %noassoc rend incorrect l'association, c'est-à-dire provoque une erreur de syntaxe si le même opérateur est trouvé deux fois "de suite".

La précédence relative entre les différents opérateurs est définie pas l'ordre dans lequel ils sont déclarés. La première déclaration **%left** ou **%right** précise les opérateurs pour lesquels la précédence est la plus faible et on peut ordonner comme ça les précédences entres opérateurs par déclaration de précédences successives.

Exemples de précédence

Pour la règle de grammaire de la figure 2.26 suivante :

```
expr:      expr MINUS expr
         | expr MULTIPLY expr
         | expr LT expr
         | LPAREN expr RPAREN
         ...
         ;
```

FIG. 2.26 – Exemple de grammaire

on va définir les règles de précédence suivantes :

Dans un exemple plus complet, qui élargirait à plusieurs autres opérateurs, on déclarerait les opérateurs par groupe de précédence. Par exemple figure 2.28 "+" est déclaré avec "-".

```
%left LT
%left MINUS
%left MULTIPLY
```

FIG. 2.27 – Définition de plusieurs précédences gauche

```
%left LT GT EQ NE LE GE
%left PLUS MINUS
%left MULTIPLY DIVIDE
```

FIG. 2.28 – Plusieurs précédences groupées

On note ici NE pour l'opérateur "not equal" et ainsi de suite pour les autres opérateurs.

Fonctionnement de la précédence

Le premier effet des déclarations de précédence est d'affecter des niveaux de précédence aux symboles terminaux déclarés. Le deuxième effet des d'affecter des niveaux de précédences à certaines règles : chaque règle obtient sa précédence du dernier symbole terminal mentionné dans ses composants.

On peut aussi spécifier explicitement la précédence d'une règle, voir la sous-section 2.5.4 sur les précédences dépendantes du contexte.

En définitive, la résolution des conflits fonctionne en comparant les précédences des règles considérées avec le token à venir. Si la précédence du token est plus élevée, le choix est de shifter. Si la précédence de la règle est plus élevée, le choix est de réduire. S'ils ont une précédence égale, le choix se fait à partir de l'associativité du niveau de précédence. Le fichier de sortie "explicite" obtenu à l'aide de l'option -v (voir la section Invoquer Ocamlyacc 2.8) comment est résolu chaque conflit.

Ni toutes les règles, ni tous les tokens n'ont de règles de précédence. Si aucune des règles ou token à venir n'ont de précédence, on shift par défaut.

2.5.4 6.4. Context-Dependent Precedence

Souvent, la précédence d'un opérateur dépend du contexte. Par exemple, un signe moins a typiquement une très forte précédence en tant qu'opérateur unaire, mais on dirait une précédence un peu moindre (moindre que la multiplication) en tant qu'opérateur binaire.

Les déclarations de précédences, **%left**, **%right** et **%nonassoc** ne peuvent être utilisées qu'une fois pour un token donné. Ainsi un token n'a qu'une seule déclaration de précédence. Afin de déclarer des précédences dépendantes du contexte, il faut utiliser un autre mécanisme, le modificateur **%prec** pour les règles.

Le modificateur **%prec** déclare la précédence d'une règle en particulier en spécifiant le symbole terminal pour lequel la précédence doit être utilisée dans cette règle. Il n'est pas nécessaire que ce symbole apparaisse ailleurs dans le règle. La syntaxe du modificateur est : **%prec** terminal−symbol et est écrite à la suite des composants de la règle. Son effet est d'affecter à la règle la précédence du symbole terminal, en écrasant la précédence qui aurait pu être déduite classiquement. La précédence de la règle altérée est utilisée pour résoudre les conflits (voir la sous-sous-section 2.3.7 sur les opérateurs de précédence).

Voici figure 2.29 comment **%prec** résoudre le problème du moins unaire. On déclare d'abord

une précédence pour le symbole terminal fictif UMINUS. Il n'y a pas de token de ce type mais le symbole sert pour marquer sa précédence :

```
%...
%left PLUS MINUS
%left MULTIPLY
%left UMINUS
....
% exp:      ...
%          | exp MINUS exp
%          ...
%          | MINUS exp %prec UMINUS
```

FIG. 2.29 – Définition des précédences de contexte

2.5.5 Etats du parser

La fonction *yyparse* est implémentée en utilisant une machine à états finis. Les valeurs "pushed" sur la pile du parseur ne sont pas simplement des codes de type de token, elles représentent des séquences entières de symboles terminaux et non-terminaux sur ou près du sommet de la pile. L'état courant accumule toutes les informations concernant les entrées précédentes qui sont significatives pour décider de ce qui doit être fait ensuite.

Chaque fois qu'un token à venir est lu, l'état courant du parser, avec le type du token à venir sont associés dans une table. La table des entrées peut indiquer "shifter le token à venir". Dans ce cas, il peut aussi spécifier le nouvel état du parseur, qui est "pushed" au sommet de la pile du parser. La table des entrées peut aussi indiquer "réduire en utilisant la règle r". Ce qui signifie qu'un certain nombre de token ou groupes sont dépilés et remplacés par un groupe. En d'autres termes, un nombre d'états est dépilé de la pile et un nouvel état est "pushed".

Il existe une autre alternative : la table peut indiquer que le token à venir est erroné dans l'état courant. Cela provoque une erreur de processing (voir la section sur la récupération d'erreur : 2.6).

2.5.6 Les conflits Reduce/Reduce

Un conflit reduce/reduce arrive s'il y a deux ou plusieurs règles qui s'appliquent à la même séquence d'entrée. Généralement, cela indique une sérieuse erreur dans la grammaire.

Par exemple figure 2.30, voici une tentative pour définir une séquence de zéro ou plus groupes de mots.

L'erreur est une ambiguïté : il y a plus d'un moyen de parser un simple mot dans une séquence. Il pourrait être réduit comme maybeword puis comme sequence à l'aide de la deuxième règle. Mais également,"rien du tout" pourrait être réduit en sequence à l'aide de la première règle et être combiné avec le mot en utilisant la 3ème règle de sequence.

Il y a également plusieurs moyens de réduire "rien du tout" en une sequence. Cela peut être fait directement à l'aide de la première règle ou indirectement à l'aide de la règle maybeword puis la 2ème règle.

On pourrait penser que cette distinction ne fait pas de différence, puisque ça ne change pas le fait qu'en entrée soit valide ou non. Mais cela affecte quelles actions peuvent être appliquées. Un ordre de parsing applique les actions de la seconde règle, l'ordre d'un autre parsing applique

```
sequence: /* empty */   { printf "empty sequence\n" }
        | maybeword     {}
        | sequence word { printf "added word %s\n" $2 }
        ;

maybeword: /* empty */  { printf "empty maybeword\n" }
        | word          { printf "single word %s\n" $1 }
        ;
```

FIG. 2.30 – Erreur de grammaire provoquant un conflit reduce/reduce

les actions de la première puis de la troisième règle. Dans cet exemple, la sortie du programme change.

Ocamlyacc résout les conflits reduce/reduce en choisissant d'utiliser les règles dans l'ordre d'apparition dans la grammaire, mais cela peut être risqué de se reposer seulement là dessus. Chaque conflit reduce/reduce doit être étudié et éliminé. Voici figure 2.31 un moyen correct de définir sequence :

```
sequence: /* empty */   { printf "empty sequence\n" }
        | sequence word         { printf "added word %s\n" $2 }
        ;
```

FIG. 2.31 – Résolution du conflit reduce/reduce

Voici figure 2.32 un autre exemple d'erreur commune qui conduit à un conflit reduce/reduce :

```
sequence: /* empty */
        | sequence words
        | sequence redirects
        ;

words:    /* empty */
        | words word
        ;

redirects:/* empty */
        | redirects redirect
        ;
```

FIG. 2.32 – Exemple 2 d'un conflit reduce/reduce

L'intention ici est de définir une séquence qui peut contenir soit des groupes word soit des groupes redirect. Les définitions individuelles de sequence, words et redirects n'ont pas d'erreurs, mais l'assemblage de toutes apporte une ambiguïté subtile : un entrée vide peut être parsée par une infinité de moyen !

Ici, "rien du tout" peut est un words, ou 2 words à la suite ou 3 ou n'importe quel nombre. "rien du tout" peut tout aussi être redirects une, deux ou n'importe quel nombre de fois. Ou words suivi de trois redirects et d'autres words...

Voici donc figure 2.33 un premier moyen de corriger ces règles, en n'ayant qu'un niveau de `sequence` :

```
sequence: /* empty */
        | sequence word
        | sequence redirect
        ;
```

FIG. 2.33 – Résolution 1 de l'exemple 2 d'un conflit reduce/reduce

Figure 2.34, on corrige en évitant soit `word` soit `redirects` d'être vide :

```
sequence: /* empty */
        | sequence words
        | sequence redirects
        ;

words:    word
        | words word
        ;

redirects:redirect
        | redirects redirect
        ;
```

FIG. 2.34 – Résolution 2 de l'exemple 2 d'un conflit reduce/reduce

2.5.7 Mystérieux Reduce/Reduce Conflits

Il arrive que les conflits reduce/reduce puissent se produire sans détection garantie. Figure 2.35 montre un exemple :

Il pourrait sembler que cette grammaire puisse être parsée avec seulement un simple token à venir : lorsque `param$_$spec` est lu, un `ID` est un `name` si "," ou " :" ou un **`type`** si un autre `ID` est à la suite. En d'autres termes, cette grammaire est LR(1).

Cependant, Ocamlyacc , comme la plupart des générateurs de parseurs, ne peut pas supporter toutes les grammaires LR(1). Dans la grammaire de cet exemple 2.35, deux contextes : après un `ID` au début d'un `param$_$spec` et `ID` au début d'un `return$_$spec`, sont suffisamment similaires pour que Ocamlyacc pensent qu'ils soient identiques. Ils apparaissent identiques à Ocamlyacc car le même ensemble de règles pourrait être activé (celle pour réduire en un `name` et celle pour réduire en un **`type`**. Ocamlyacc ne peut pas déterminer à ce moment de la génération du parseur, que les règles nécessiteront un token à venir différent dans les deux contextes, il construit donc un seul état de parser pour eux. Or, combiner les deux contextes provoque un conflit ultérieurement. En terme de parser, ce cas illustre qu'il ne s'agit pas d'une grammaire LALR(1).

Il est plutôt complexe de corriger Ocamlyacc pour lui permettre de parser des grammaires LR(1) qui ne sont pas LALR(1). Les générateurs de parseur qui le font ont tendance à générer des parseurs très très gros. En pratique, on préfère laisser Ocamlyacc tel qu'il est.

```
%token ID COMMA COLON

%%
def:     param_spec return_spec COMMA
         ;
param_spec:
               type
         |     name_list COLON type
         ;
return_spec:
               type
         |     name COLON type
         ;
type:          ID
         ;
name:          ID
         ;
name_list:
               name
         |     name COMMA name_list
         ;
```

FIG. 2.35 – Conflit reduce/reduce non détectable

Lorsque le problème LR(1) se présente, on peut souvent le résoudre en identifiant les deux états de parseur qui prêtent à confusions et ajouter quelque chose qui les feront se distinguer. Dans l'exemple précédent, ajouter une règle à `return$_$spec` comme dans le figure 2.36 permet de résoudre le problème :

```
%token BOGUS
...
%%
...
return_spec:
               type
         |     name COMMA type
         /* This rule is never used.  */
         |     ID BOGUS
         ;
```

FIG. 2.36 – Résolution d'un conflit reduce/reduce non détectable

Cela corrige le problème en introduisant la possibilité d'activer une règle supplémentaire dans le contexte après `ID` au début du `return$_$spec`. Cette règle n'est pas active dans le contexte correspondant de `param_spec`, et de cette façon les deux contextes reçoivent des états de parseur différents. Du moment que le token `BOGUS` n'est pas généré par Ocamllex , la règle ajoutée ne modifie pas la façon de parser l'entrée.

Dans cet exemple particulier, il y a un autre moyen de résoudre le problème : réécrire la

règle de `return$_$spec` afin d'utiliser `ID` directement, autrement que via `name`. Cela permet que les deux contextes aient des ensembles de règles actives différents, puisque la règle pour `return$_$spec` active la règle modifiée pour `return$_$spec` plutôt que celle pour `name`. Ceci est présenté figure 2.37 :

```
param_spec:
            type
        |   name_list COMMA type
        ;
return_spec:
            type
        |   ID COMMA type
        ;
```

FIG. 2.37 – Résolution d'un conflit reduce/reduce non détectable - 2ème possibilité

2.6 Error récupération

En général, il ne faut pas qu'un programme s'interrompre sur une erreur de parsing. Par exemple, un compilateur devrait résister aux erreurs suffisamment pour parser le reste du fichier d'entrée à la recherche d'erreurs ; une calculatrice devrait accepter une autre expression.

Lorsque l'on fait du parsing avec une simple commande interactive où chaque entrée est constitué d'une seule ligne, il peut être suffisant d'avoir l'appelant qui attrape l'exception et qui ignore le reste de la ligne d'entrée lorsqu'une erreur survient (et on appelle la fonction de parsing à nouveau pour continuer sur une autre ligne). Mais cela n'est pas adapté pour un compilateur, parce qu'il "oublie" tout le contexte syntaxique qui a amené à l'erreur. Une erreur de syntaxe dans un fonction du compilateur ne devrait pas provoquer que la "ligne" suivante soit traitée de la même manière que si c'était le début du fichier source.

On peut donc définir comment résister à une erreur de syntaxe en écrivant des règles qui reconnaissent un token spécial d'erreur. C'est un symbole terminal qui est réservé à la récupération d'erreur. Le parseur Ocamlyacc génère un token d'erreur lorsque se produit une erreur de syntaxe ; si on définit une règle qui reconnaît ce token dans le contexte, le parsing peut alors continuer.

Par exemple, figure 2.38 :

```
stmnts:  /* empty string */              {}
       | stmnts NEWLINE                  {}
       | stmnts exp NEWLINE     {}
       | stmnts error NEWLINE   {}
```

FIG. 2.38 – Rattrapage d'erreur

La quatrième règle dans cet exemple dit qu'`error` suivie d'une nouvelle ligne `NEWLINE` sera une combinaison valide à n'importe quel `stmnts`.

Que se passe-t-il si une erreur de syntaxe se produit au milieu d'une `exp` ? La règle de récupération d'erreur, interprétée strictement, va appliquer la séquence précise des `stmnts`, une `error` et une `NEWLINE`. Si une erreur se produit au milieu d'une `exp`, il y aura probablement

des tokens supplémentaires et des sous-expressions dans la pile après le dernier `stmnts`, et il y aura des tokens à lire avant la prochaine `NEWLINE`. La règle n'est donc pas applicable normalement.

Mais Ocamlyacc peut forcer la situation de façon à correspondre à la règle, en éliminant une partie du contexte sémantique et une partie de l'entrée. Premièrement, Ocamlyacc élimine les états et les objets de la pile jusqu'à revenir à l'état dans lequel le token `error` est accepté. (Cela signifie que les sous-expressions déjà parsées sont éliminées, en retournant au dernier `stmnts` complet.) A ce niveau, le token `error` peut être shifté. Alors, si l'ancien token à venir n'est pas acceptable pour être shifté en suivant, le parseur lit les tokens et les éliminent jusqu'à trouver un token qui est acceptable. Dans cet exemple 2.38, Ocamlyacc lit et élimine les entrées jusqu'à la prochaine `NEWLINE` afin que la quatrième règle puisse s'appliquer.

Le choix des règles d'erreur dans une grammaire est un choix de stratégie pour la récupération d'erreur. Une stratégie simple et utile est simplement de skiper le reste de la ligne d'entrée courante ou d'état courant si une erreur est détectée, comme le montre l'exemple de la figure 2.39 :

```
stmnt: error SEMICOLON  {}      /* en cas d'erreur, skiper jusqu'au ;
    */
```

FIG. 2.39 – Stratégie de récupération d'erreur

Il est aussi utile de récupérer jusqu'au délimiteur-fermant correspondant à un délimiteur-ouvrant qui aurait déjà été parsé. Sinon, le délimiteur-fermant apparaîtra probablement isolé et générera un autre message d'erreur (erroné), voir figure 2.40 :

```
primary:  LPAREN expr RPAREN    {}
        | LPAREN error RPAREN   {}
        ...
        ;
```

FIG. 2.40 – Erreur causé entre deux délimiteurs

Les stratégies de récupération d'erreur sont nécessairement des tentatives. Lorsqu'une tentative échoue, une erreur de syntaxe mène souvent à une autre... Dans l'exemple précédent, la règle de récupération d'erreur suppose qu'une erreur est due à une mauvaise entrée à l'intérieur d'un `stmnts`. Supposons qu'un `$;$` soit inséré au milieu d'un `stmnts` valide. Après avoir utilisé les règles de récupération d'erreur de la première erreur, une autre erreur de syntaxe est trouvée ensuite : puisque le texte qui suit le `$;$` erroné est également un `stmnts` invalide.

Afin d'éviter les messages d'erreur abondants, le parseur ne générera pas de message d'erreur pour une erreur de syntaxe qui se produit juste après la première ; seulement après trois token d'entrée consécutifs correctement shifté, les messages d'erreur sont à nouveau générés le cas échéant.

2.7 Debugger son parser

Pour débugger le parser généré par Ocamlyacc :

– En utilisant la commande `ocamlyacc -v filename.mly`, on génère les informations de parsing dans un fichier appelé filename.output. Ces informations consistent en une table de parsing et des indications concernant les conflits.

– Mettre l'option p à la variable d'environnement OCAMLRUNPARAM
(`export OCAMLRUNPARAM='p'` dans un shell bash.

Le parseur affiche des messages à propos de ses actions telles que shifter un token, réduire une règle...

On peut obtenir le numéros de la règle ou les numéros d'états mentionnés dans les messages dans filename.output.

2.8 Executer Ocamlyacc

La manière habituelle d'invoquer ocamllyacc : `ocamlyacc filename.mly` . Où filename.mly est le nom du fichier de grammaire. Le nom du fichier de parsing est obtenu en remplaçant le .mly par .ml. Ocamlyacc génère un .ml à partir du .mly

2.8.1 Ocamlyacc Options

Voici une liste d'options qui peuvent être utilisées avec Ocamlyacc :

– `−v` Par défaut, cette option génère un fichier filename.output. Il contient des informations de parsing telle que la description des tables de parsing et un rapport des ambiguïtés de la grammaire.

– `−bfname` Permet de changer le nom du fichier de sortie par fname.ml fname.mli et fname.output

Chapitre 3

Astuces

- Quand on fait du parsing "en ligne" plutôt qu'à l'aide d'un fichier, CTRL D pour fin de fichier
- Utiliser < et > pour entrée et sortie du programme complet : ./prog < entree > sortie
- Pour vérifier la grammaire, lire .output
- Faire un Makefile quand on travaille sur le processus complet (analyse lexicale, syntaxique et actions sémantiques...). Voir l'exemple figure 3.1.

```
.PRECIOUS: %.mli

all: prog

prog:  lexer.cmo parser.cmo prog.cmo
	ocamlc -o $@ $^

%.cmo: %.ml
	ocamlc -c $<

%.cmi: %.mli
	ocamlc -c $<

%.ml: %.mll
	ocamllex $<

%.ml %.mli: %.mly
	ocamlyacc -v $<


lexer.cmo: parser.cmi

clean:
	rm -f *.cmi *.cmo
```

FIG. 3.1 – Exemple de Makefile