Interpréter un petit langage impératif

Thibault Suzanne

5 juin 2012

Table des matières

1 Introduction **1**

2 Prérequis **1**

3 Commençons ! **1**
3.1 Comment nous fonctionnerons . . . 1
3.2 Notre langage . . . 2
3.3 Les différentes étapes de l'interprétation . . . 3

4 L'analyse lexicale **3**
4.1 Mots, lexèmes et découpage . . . 3
4.2 Les expressions rationnelles . . . 4
4.3 Ocamllex . . . 6
4.3.1 Le commencement . . . 6
4.3.2 Les mots-clefs . . . 7
4.3.3 Les nombres, la première loi et les alias . . . 8
4.3.4 Les variables et la deuxième loi . . . 9
4.3.5 Les caractères blancs . . . 10
4.3.6 Les commentaires : une nouvelle règle . . . 11
4.3.7 Utilisation pratique de notre analyseur . . . 13

5 L'analyse syntaxique **16**
5.1 Arbres syntaxiques . . . 16
5.2 Les grammaires hors-contexte . . . 20
5.3 Menhir . . . 22
5.3.1 Présentation . . . 22
5.3.2 Installation . . . 22
5.3.3 Redéfinissons nos lexèmes . . . 23
5.3.4 Passons aux choses sérieuses . . . 23
5.3.5 Où l'on termine l'écriture de notre AST avec les « commandes » du langage . . . 26
5.3.6 Compilons ! . . . 28

5.3.7 Au secours, ça ne marche pas ! 29

1 Introduction

Vous avez appris un langage de programmation, vous avez écrit plein de jolies choses avec. Une fois vos codes écrits, vous utilisiez un programme externe pour les exécuter (ou les compiler puis les exécuter). Vous avez même peut-être travaillé avec succès sur des gros projets qui ont fait appel à toutes vos connaissances de votre langage favori. Tout cela est très bien. Mais savez-vous comment ce programme externe avait lui-même été créé ?

Si vous n'en avez aucune idée, ou que vous êtes tout simplement curieux de le savoir, vous êtes au bon endroit. Cet article vous montrera en détail comment interpréter un petit langage impératif, sous forme de cours-TP où nous mettrons cela en place ensemble.

2 Prérequis

Interpréter un langage n'est pas quelque chose de très compliqué (surtout si le langage est suffisamment simple), mais vous devez tout de même savoir certaines choses. Concrètement, vous serez capable de lire et de comprendre ce cours si vous maîtrisez un minimum le langage OCaml. Notamment, vous devez être bien à l'aise avec les types récursifs (définition et utilisation), et savoir appliquer ces connaissances à la manipulation d'arbres. Vous devez également connaître les bases des entrées/sorties (notamment la lecture dans un fichier). Enfin, il est plus que recommandé de connaître les bases d'un langage impératif, par exemple le C ou le Python.

3 Commençons !

3.1 Comment nous fonctionnerons

Ce « cours » est écrit sous la forme d'un TP : en le parcourant, nous construirons progressivement notre interpréteur. Les explications pratiques du fonctionnement des outils se font beaucoup à partir du code, et vous avez à la fin de chaque partie la source complète du fichier final que nous aurons écrit. Cela veut dire que, bien sûr, vous pouvez vous contenter de faire un copier-coller des différents fichiers et des commandes pour arriver à obtenir un interpréteur qui fonctionne. Ce serait cependant dommage pour vous : je vous encourage à essayer au maximum de coder par vous-mêmes, et à vous servir du corrigé quand vous n'arrivez pas à comprendre quelque chose ou que vous voulez comparer votre solution. Bref, il est là volontairement, mais vous apprendrez plus si vous le copiez le moins possible.

3.2 Notre langage

Avant de commencer à implémenter notre langage, nous allons choisir à quoi il va ressembler. Nous ferons quelque chose de simple : nous allons écrire un interpréteur pour un langage impératif minimaliste.

Voici donc les éléments de base de notre langage :

– les entiers, qui seront soit écrits directement, soit contenus dans des variables

- les opérations +, - et * sur les entiers
- les booléens, incluant true, false et la comparaison > sur les entiers
- les opérations and, or et not sur les booléens
- les boucles while
- les conditions if
- une instruction print pour afficher un entier

Les opérations sur les entiers et les booléens seront régies par les règles de priorité usuelles. Pour agir sur ces priorités, on pourra utiliser les parenthèses. Pour affecter une valeur à une variable, on utilisera := (la création de la variable sera effectuée à sa première affectation, comme en Python par exemple).

Et, comme tout langage qui se respecte (et le nôtre se respecte particulièrement, s'il vous plaît !), nous aurons des commentaires : // pour un commentaire sur une ligne, et /* ... */ pour des commentaires multi-lignes.

Voilà un exemple de code pour que vous visualisiez bien la syntaxe (qui est très simple) :

```
// On initialise les variables
n := 5
f := 1

/*
  boucle principale
  /* commentaire imbriqué */
*/

while n > 0 {
    f := f * n
    n := n - 1
    }


print f
/* On peut même print-er des expressions numériques */
print 3 + 4 * 5
```

3.3 Les différentes étapes de l'interprétation

Pour comprendre les différentes étapes de l'interprétation d'un programme, il est souvent pratique de faire la comparaison avec la langue que vous parlez tous les jours. Imaginez que vous discutiez avec un ami, et qu'il vous dise « Le français est une belle langue ». Si vous, vous comprenez tout de suite cette phrase, ce n'est pas le cas de votre ordinateur. Voilà les étapes qu'il suivra pour en déchiffrer le sens :

- Initialement, il ne connaît qu'une suite de caractères. Parallèlement, c'est comme si vous en étiez au stade où vous avez compris : « leufranssèhètunebèlelang ».
- La première étape est **l'analyse lexicale** : la phrase sera découpée en une suite de mots connus. Vous savez alors que votre ami a prononcé successivement les mots « le », « français », « est », « une », « belle » et « langue ». Vous connaissez alors la nature de chaque mot individuellement (« le » est un article défini, « français » est un substantif...), mais vous ne savez pas encore comment ils sont reliés

entre eux.

- La deuxième étape est **l'analyse syntaxique**. À partir de votre suite de mots, vous construisez des phrases. Vous savez maintenant que « le français » est un groupe nominal, et qu'il est sujet du verbe « est ». Mais vous ne savez toujours pas quel sens a cette phrase.
- La dernière étape est **l'analyse sémantique**. Maintenant que vous connaissez la structure de la phrase, vous pouvez comprendre ce qu'elle veut dire. En puisant dans votre vocabulaire, vous savez maintenant que votre ami vous a indiqué que la langue que vous parlez, le français, a des caractéristiques qui la rendent tout à fait intéressante.

Votre interpréteur fonctionnera (presque) exactement de cette façon. Si vous ne voyez pas encore parfaitement comment tout cela va s'agencer, ce n'est pas grave : vous comprendrez vraiment une fois quue nous aurons réalisé chacune de ces composantes. Il est simplement important de retenir que l'interprétation se fera en 3 étapes, qui ont chacune un rôle bien précis et qui agissent sur le résultat de l'étape précédente.

4 L'analyse lexicale

4.1 Mots, lexèmes et découpage

Il est maintenant temps de commencer la réalisation de notre interpréteur. Comme nous l'avons vu plus haut, la première étape est celle de l'analyse lexicale, qui consiste à découper notre code source en « mots » de notre langage.

Plus exactement, l'analyseur lexical (on entend souvent le terme anglais *lexer*) va prendre en entrée une chaîne de caractères, qui sera notre code source, et produire en sortie une suite de « lexèmes » (les anglophones vous parleront de *tokens*), qui seront les « mots » ou « unités lexicales » de notre langage.

Les lexèmes seront tout simplement définis par un type énuméré :

```
type token =
  | True
  | False
  | Not
  | And
  | Or
  | Greater
  | Equal
  | LeftPar
  | RightPar
  | LeftCurly    (* { *)
  | RightCurly   (* } *)
  | Affect       (* := *)
  | If
  | While
  | Print
  | Plus
  | Times
  | Minus
  | Int of int
```

```
21    | Var of string
22    | Eof          (* Fin de fichier *)
```

Par exemple, au code suivant :

```
1  while n > 0 {
2      n := n - 1
3      }
```

Correspondra la liste de lexèmes suivantes :

```
1  [While; Var "n"; Greater; Int 0; LeftCurly; Var "n"; Affect; Var "n";↩
       Minus; Int 1; RightCurly; Eof]
```

Remarquez qu'on n'a pas défini de lexème pour nos commentaires : en effet, on les retirera du code à traiter dès l'analyse lexicale, pour ne plus avoir à s'en préoccuper après.

Nous connaissons donc l'ensemble de nos lexèmes. Il nous faut maintenant écrire les correspondances avec les chaînes de caractères de notre code source. On pourrait imaginer écrire une simple fonction, prenant comme argument un chaîne de caractères, et renvoyant le lexème correspondant. Pour certains, ça fonctionnerait très bien : par exemple "while" se convertit très simplement en While. Mais comment ferait-on pour les variables par exemple ? On ne peut évidemment pas écrire un cas pour chaque nom de variable possible, puisqu'il y en a une infinité... En fait, il faut trouver un moyen de décrire les chaînes de caractères « qui sont une suite de lettres ou de chiffres commençant par une lettre ». Idem pour les nombres : un entier, c'est « une suite de chiffres ».

Alors bien sûr, on pourrait encore une fois écrire une fonction qui indique si une chaîne de caractère correspond bien au motif d'une variable. Ça marcherait plus ou moins, et on pourrait s'en tirer honorablement pour trouver le lexème correspondant à un bout de chaîne donné. Mais on n'est pas plus avancé : comment savoir quelle partie du code source fournir à cette fonction pour déterminer le lexème en question ? Si mon code est resultat := resultat + 4, comment décider si le prochain lexème à analyser correspondra à resultat, r ou resultat : ?

4.2 Les expressions rationnelles

La solution à ce double problème existe environ depuis les années 1940 (!!), et elle s'appelle « expressions rationnelles ». Vous en avez peut-être déjà entendu parler (on les appelle parfois « expressions régulières », ou « regex » en anglais) : c'est à la fois un objet théorique intéressant à étudier, et un outil pratique qui est très puissant lorsqu'on s'en sert correctement mais que les programmeurs d'aujourd'hui ont tendance à utiliser pour tout et n'importe quoi. Du coup, on en retrouve beaucoup de versions mutantes à des endroits où elles ne devraient pas être et où elles compliquent beaucoup le code et sa maintenance, et c'est probablement ce à quoi elles doivent leur réputation assez sulfureuse auprès de nombreux programmeurs.

La théorie des expressions rationnelles n'est pas spécialement compliquée mais est trop longue pour entrer entièrement dans cet article. J'en ferai tout de même une présentation rapide, mais je vous encourage vivement à vous renseigner dessus par vous-mêmes, ne serait-ce que parce que ça fait partie du bagage culturel que vous devriez posséder.

Pour ceux qui connaissent déjà les expressions régulières, il est important que vous lisiez quand même cette partie : la syntaxe que nous utiliserons diffère légèrement des syntaxes « classiques » que vous

connaissez peut-être (qu'on rencontre par exemple avec Python, Perl ou sed).

La première chose que vous devrez retenir concernant les expressions régulières, c'est qu'une expression régulière est un « objet » informatique (dans le sens « un truc », ça n'a pas de rapport direct avec la programmation objet) qui va servir à « décrire » des chaînes de caractères qui correspondent à un certain « motif ».

L'expression régulière la plus simple a justement la forme d'une chaîne de caractères, et elle décrit la chaîne correspondante. Par exemple, l'expression régulière "we don't need no thought control" décrira la chaîne de caractères "we don't need no thought control".

Ces expressions-chaînes de caractères suivent les mêmes règles que les chaînes de caractères d'OCaml. Par exemple, l'expression "\n" décrira une chaîne de caractères ne contenant qu'un retour à la ligne. J'attire ici l'attention de ceux qui ont déjà joué avec des expressions régulières : "a*", par exemple, décrira la chaîne de caractères "a*", et pas par exemple la chaîne "aaaaa" (si vous ne comprenez pas pourquoi je dis ça, ignorez simplement cette phrase pour l'instant).

Si votre expression régulière-chaîne de caractères ne contient qu'un seul caractère, vous pouvez l'écrire en suivant la syntaxe du type char d'OCaml : ainsi, 'a' décrira la chaîne "a".

Si vous souhaitez décrire plusieurs chaînes différentes, vous pourrez utiliser un caractère spécial des expressions régulières : le tube |. Il s'interpose simplement entre deux expressions régulières, et l'expression globale obtenue décrit toutes les chaînes décrites soit par la première, soit par la seconde sous-expression. Ainsi, l'expression régulière "mer" | "montagne" décrira les chaînes de caractères "mer" et "montagne". Vous pouvez regrouper ainsi plusieurs expressions : "mer" | "montagne" | "lagon" décrira les 3 chaînes de caractères que vous devinez.

Vous pouvez coller deux expression régulières l'une après l'autre : "côté " ("mer" | "montagne") décrira les deux chaînes "côté mer" et "côté montagne". Vous aurez remarqué l'usage des parenthèses pour gérer les priorités : la concaténation de deux expressions régulières est prioritaire sur l'union |.

Le deuxième caractère spécial est l'étoile * (non, elle ne s'appelle pas astérisque quand on parle d'expressions rationnelles). Elle s'appose après une expression régulière, pour décrire 0 ou plusieurs occurrences de cette expression. Ainsi, "na"* décrira les chaînes "", "na", "nananana"... Vous pouvez bien sûr combiner plusieurs des constructions que nous avons vues : "na"* " "("bat" | "spider") "man" décrira les chaînes "nanananana batman", "na spiderman"...

Le caractère spécial + fonctionne de la même façon que l'étoile, mais sert à faire correspondre une ou plusieurs occurrences de l'expression à laquelle il est appliqué (autrement dit, il ne permet pas de décrire la chaîne vide ""). Le caractère ? fait lui correspondre 0 ou 1 occurrence.

Un concept important pour nos expressions régulières est celui de « classe ». Une classe est un ensemble d'expressions-caractères (par exemple, 'a') entre crochets, éventuellement composée aussi de nouveaux symboles spéciaux. Sans ceux-ci, c'est une autre façon d'écrire une union : ['a' 'e' 'i' 'o' 'u' 'y'] est équivalent à ("a" | "e" | "i" | "o" | "u" | "y").

On peut utiliser le caractère spécial ^ juste après le crochet ouvrant pour faire la négation d'une classe : [^ '\n'] décrira toutes les chaînes, sauf celles composées uniquement d'un retour à la ligne.

Enfin, on peut utiliser le caractère tiret - pour décrire une série de caractères « compris » entre deux : ['a' - 'e'] décrira "a", "b", "c", "d" et "e".

Une fois que vous saurez que le caractère spécial _ peut servir, un peu comme dans les match d'OCaml, à décrire n'importe quelle chaîne, vous connaîtrez tout des expressions régulières (enfin, non, ce n'est pas vrai. Mais vous en connaîtrez suffisamment pour ce qui nous concerne. Je vous engage toutefois vivement à vous renseigner un peu plus en détail dessus).

4.3 Ocamllex

4.3.1 Le commencement

Nous avons maintenant notre outil théorique qui nous permettra de découper notre code en unités lexicales. Reste à savoir comment l'utiliser en pratique. Bien sûr, nous pourrions implémenter nous-mêmes notre moteur d'expressions régulières et l'analyseur syntaxique qui va avec. C'est d'ailleurs un travail intéressant que je vous encourage à faire quand vous maîtriserez le sujet.

L'ennui, c'est que ça demande beaucoup de travail alors qu'il y a des outils tout prêts. Celui que nous utiliserons pour nous simplifier la vie s'appelle **ocamllex**.

C'est un « générateur d'analyseur lexical » : à partir d'un ensemble de règles associant des expressions régulières aux lexèmes qu'ils décrivent, il génère une fonction OCaml qui permet, à partir d'un code source, de générer la suite de lexèmes correspondant.

Ocamllex est fourni avec l'installation standard d'OCaml, pas besoin donc d'installation supplémentaire. Vous pouvez vérifier qu'il est bien disponible sur votre machine en tapant la commande `ocamllex`, qui devrait vous afficher la liste des options disponibles.

Avant de commencer, reprenez le type énuméré des lexèmes que nous avons défini plus haut, et écrivez-le dans un fichier `tokens.ml`. Pour rappel, ce type était :

```
type token =
  | True
  | False
  | Not
  | And
  | Or
  | Greater
  | Equal
  | LeftPar
  | RightPar
  | LeftCurly    (* { *)
  | RightCurly   (* } *)
  | Affect       (* := *)
  | If
  | While
  | Print
  | Plus
  | Times
  | Minus
  | Int of int
  | Var of string
  | Eof          (* Fin de fichier *)
```

Notre analyseur lexical se servira de ce fichier pour connaître les différents lexèmes qu'il est susceptible de produire. Par convention, notre code ocamllex s'écrit dans un fichier *.mll. Créez donc un fichier que vous appellerez par exemple « lexer.mll ».

Il est maintenant temps d'écrire notre analyseur. Quand vous écrivez du code OCaml, l'élément principal est une fonction :

```
let ma_fonction = function
  | antécédent1 -> résultat1
  | antécédent2 -> résultat2
```

Le code Ocamllex ressemble un peu à ça. L'élément principal s'appelle une règle, les antécédents sont des expressions régulières et les résultats sont les lexèmes correspondants. La syntaxe est la suivante :

```
rule ma_règle = parse
  | expression1 { Lexeme1 }
  | expression2 { Lexeme2 }
```

À partir de ces différentes correspondances, ocamllex construira une fonction OCaml (dont nous verrons le type exact plus tard) qui, à partir d'un code donné, renvoie le lexème suivant (et retire les caractères correspondants du code). Pour qu'il puisse connaître les lexèmes, il faut que le type ait été déclaré quelque part, comme dans un fichier OCaml classique. Pour cela, OCamllex permet d'écrire du code OCaml en tête de fichier, entre accolades.

4.3.2 Les mots-clefs

Voyons ça tout de suite en pratique avec la règle qui analyse tous les mots réservés de notre langage :

```
{
  open Tokens
  exception LexingError
}

rule lexer = parse
  | eof { Eof }
  | "true" { True }
  | "false" { False }
  | "not" { Not }
  | "and" { And }
  | "or" { Or }
  | "if" { If }
  | "while" { While }
  | "print" { Print }
  | "(" { LeftPar }
  | ")" { RightPar }
  | "{" { LeftCurly }
  | "}" { RightCurly }
  | "+" { Plus }
  | "*" { Times }
  | "-" { Minus }
  | ">" { Greater }
  | "=" { Equal }
  | ":=" { Affect }
```

Vous remarquerez qu'on ouvre en tête le fichier le module Tokens, qui correspond donc au fichier tokens.ml écrit plus haut. On définit également une exception LexingError qui nous servira plus tard en cas d'erreur lexicale.

En plus des quelques mots réservés de notre langage, vous avez remarqué la présence de l'antécédent eof. C'est un mot-clef OCamllex qui indique la fin de fichier : il est important de renvoyer le lexème correspondant, que nous avons appelé Eof, sinon vous aurez des problèmes lors de la phase d'analyse syntaxique (qui ne saura pas bien s'arrêter). Le reste du code est assez simple pour être compris sans commentaires.

4.3.3 Les nombres, la première loi et les alias

Passons maintenant aux nombres. Comme nous l'avons vu plus haut, nous ne gèrerons que des nombres entiers, qui seront donc une suite d'un ou plusieurs chiffres, un chiffre étant compris entre '0' et '9' pour l'ordre des caractères. Ça tombe bien, nous savons décrire ces caractères, et nous savons aussi comment exprimer "1 ou plusieurs". L'expression régulière permettant de décrire nos nombres est donc, si vous avez bien suivi, ['0' - '9']+. Il nous reste donc à écrire le lexème produit. Pour cela, OCamllex nous permet de désigner la chaîne de caractères correspondante à l'aide du mot-clef as. Nous écrirons donc :

```
rule lexer = parse
  | ...
  | ['0' - '9']+ as n { Int (int_of_string n) }
```

Notez donc l'utilisation du as, et n'oubliez pas de convertir la chaîne n en entier : notre lexème Int attend en effet un paramètre de type int.

Un premier problème se pose ici : notre analyseur reconnaît, pour les nombres, une suite de 1 ou plusieurs chiffres. Comment savoir où il va s'arrêter ? Par exemple, si notre code contient a := 123 et qu'il en est au 123, comment saura-t-il s'il doit traduire 1, 12 ou 123 ?

La solution à ce problème est apportée par la loi de la plus longue correspondance. C'est la première loi que suivra notre analyseur en cas d'ambiguité : comme son nom l'indique, elle précise que la chaîne de caractères correspondant au lexème produit à une certaine étape est toujours la plus longue que l'anayseur lexical peut reconnaître à cette étape. Ainsi, dans l'exemple précédent, la chaîne reconnue était 123.

Cette loi est l'occasion pour nous de parler un (tout) petit peu du fonctionnement interne de notre analyse lexicale. À l'aide d'objets informatiques appelés automates finis, qui sont très fortement liés aux expressions rationnelles, l'analyseur parcourt chaque caractère de la chaîne jusqu'à ce que la sous-chaîne entre le début de la chaîne principale et le caractère courant ne puisse plus constituer le début d'une chaîne reconnue par l'automate. L'analyseur renvoie alors la dernière chaîne rencontrée qu'il reconnaissait comme correspondant à un lexème.

Nous allons maintenant introduire une autre possibilité offerte par OCamllex : les alias. Ils permettent de nommer des expressions régulières afin de s'en resservir par la suite sans avoir à les réécrire à chaque fois. En plus du gain de place pour des grosses expressions souvent réutilisées, ils permettent souvent d'améliorer la lisibilité. Ici, nous allons définir un alias « chiffre ». On utilise pour cela le mot-clef let, comme en OCaml, et on déclare l'alias avant son utilisation (donc ici entre le code OCaml initial et notre règle). Voilà donc l'état actuel de notre fichier :

```
{
  open Parser
  exception LexingError
```

```
}

let digits = ['0' - '9']

rule lexer = parse
  | eof { Eof }
  | "true" { True }
  | "false" { False }
  | "not" { Not }
  | "and" { And }
  | "or" { Or }
  | "if" { If }
  | "while" { While }
  | "print" { Print }
  | "(" { LeftPar }
  | ")" { RightPar }
  | "{" { LeftCurly }
  | "}" { RightCurly }
  | "+" { Plus }
  | "*" { Times }
  | "-" { Minus }
  | ">" { Greater }
  | "=" { Equal }
  | ":=" { Affect }
  | digits+ as n { Int (int_of_string n) }
```

Rien de très compliqué.

4.3.4 Les variables et la deuxième loi

Nous allons maintenant passer aux variables. Elles doivent commencer par une lettre (pour faire pompeux, on peut dire un « caractère alphabétique »), éventuellement suivie d'un ou plusieurs « caractères alphanumériques » (des chiffres ou des lettres, appellation qu'on rencontre plus souvent). Les lettres peuvent être en majuscule. Comme tout à l'heure, nous définissons un nouvel alias avant d'écrire notre expression régulière :

```
let digits = ['0' - '9']
let alpha = ['a' - 'z' 'A' - 'Z']

rule lexer = parse
  | "true" { True }
  | alpha (alpha | digits)* as v { Var v }
```

Les plus malins auront remarqué que nous avons ici un deuxième problème. En effet, le mot "true" est lui-même reconnu à la fois par l'expression régulière "true", afin de produire le lexème attendu True des booléens, mais aussi par l'expression décrivant les variables.

On pourrait se débrouiller pour que l'expression des variables ne décrive pas "true", mais on serait obligés de faire ça pour chaque mot-clef : ce serait un travail long, compliqué, délicat, stupide, *bogue-amical* et déraisonnable. Si vous n'êtes pas convaincus, essayez vous-mêmes, et une semaine après, rajoutez un mot-clef à votre langage.

Comme tout-à-l'heure avec la plus longue correspondance, une deuxième loi existe pour régler ce problème : la priorité à l'expression la plus haute. Elle indique qu'en cas de conflit pour une chaîne donnée, éventuellement après application de la loi de la plus longue correspondance, qui permettra par exemple de ne pas avoir de conflit sur la variable "truefalse", le lexème produit est le premier placé dans la liste des correspondances du fichier OCamllex. Ainsi, en plaçant les mots-clefs avant les variables dans ce fichier, on est assuré qu'ils seront bien reconnus comme tels.

4.3.5 Les caractères blancs

La prochaine étape est la lecture des caractères blancs. Comme le montre le code-exemple du début de ce tutoriel, l'utilisateur de notre langage a la possibilité d'utiliser des caractères tels que l'espace, la tabulation ou le saut de ligne pour rendre son code plus agréable à la lecture. Même s'ils ne correspondent à aucun lexème, nous sommes obligés de les noter quelque part dans notre code OCamllex, sinon il s'arrêterait dès qu'il en rencontre un avec une erreur lexicale.

Pour cela, nous utiliserons le mot clef lexbuf. Il désigne, après reconnaissance d'un mot dans le code (correspondant donc à une expression régulière), la suite des caractères de ce code. Lorsque nous rencontrons un caractère blanc, il nous suffit donc de réappeler notre lexer (qui, rappelez-vous, doit renvoyer à chaque appel le prochain lexème du code) sur ce lexbuf afin qu'il y trouve le lexème suivant. Les règles écrites en OCamllex sont automatiquement récursives, nous écrirons donc :

```
1 (* On ne prendra en compte que les sauts de lignes, les tabulations ↩
     et
2 les espaces *)
3
4 let empty = ['\n' '\t' ' ']
5
6 rule lexer = parse
7   | empty+ { lexer lexbuf }
8   | ...
```

Notez que nous avons choisi de sauter d'un coup tous les caractères blancs consécutifs (à l'aide du signe +), mais qu'on aurait très bien pu le faire un par un.

4.3.6 Les commentaires : une nouvelle règle

Avant d'avoir un analyseur lexical complet, il ne nous reste plus qu'à gérer les commentaires. Rappelez-vous qu'ils ont la syntaxe des commentaires C : // pour commenter sur une ligne (ou une fin de ligne), et /* ... */ pour délimiter des commentaires pouvant s'étendre sur plusieurs lignes.

Pour les premiers, c'est relativement simple : il nous suffit d'ignorer (comme avec les blancs) tous les caractères compris entre // et la fin de ligne '\n'. Ces caractères peuvent être n'importe quoi... sauf eux-mêmes des sauts de lignes ! (Un bonbon pour ceux qui avaient deviné.) Je vous laisse écrire l'expression correspondante par vous-mêmes, la solution viendra plus bas avec le code OCamllex complet.

Pour les commentaires multilignes, c'est un peu plus compliqué. En effet, nous ne pouvons pas nous contenter de faire correspondre un /* avec le */ suivant : nous voulons pouvoir imbriquer des commentaires, et ce à n'importe quel niveau. Par exemple, on veut pouvoir écrire :

```
/*
  Premier niveau de commentaires
  /*
    Deuxième niveau
    /* Troisième niveau */
  */
*/
```

Inutile d'essayer d'écrire l'expression régulière permettant de décrire ces commentaires imbriqués, vous n'y arriverez pas pour la bonne et simple raison qu'elle n'existe pas. Si vous voulez connaître le mot pour briller en société, on dit que ce langage des commentaires imbriqués est « non-régulier » (ce qui correspond exactement à « non reconnaissable par une expression rationnelle/un automate » - les deux revenant au même). Nous allons donc devoir tricher un peu pour reconnaître plus de choses qu'il n'en est théoriquement possible avec nos expressions régulières.

Pour cela, nous allons écrire une deuxième règle, qui sera appelée dès que l'on rencontrera un début de commentaire multiligne /*. Elle s'occupera d'ignorer tous les caractères jusqu'au */ correspondant, puis repassera la main à notre règle lexer. Afin de trouver la fermeture de commentaire correspondante, elle devra connaître le niveau de commentaire dans lequel on se trouve : si on en est au niveau n et qu'on arrive à un mot /*, on passe au niveau $n + 1$. Si on trouve */, on passe au niveau $n - 1$. Si on était alors au premier niveau, il suffit d'appeler la règle lexer sur la suite du code : on a alors ignoré le commentaire comme prévu.

La définition de cette nouvelle règle se fera avec le mot-clef and, qui encore une fois ressemble beaucoup dans son fonctionnement à celui d'OCaml. La mémorisation du niveau de commentaire pourrait se faire avec une référence utilisée dans le code OCaml des résultats (et correctement définie en tête de fichier), je vous invite d'ailleurs à le faire à titre d'exercice. Pour ce projet, j'utiliserai plutôt un argument supplémentaire passé à la règle, correspondant à la profondeur des imbrications au moment où cette règle est appelée. Lorsqu'on rencontre un /* dans lexer, on appelle donc la règle comment avec l'argument 1 (sans oublier l'argument lexbuf).

Le corps de cette règle devrait être assez simple pour vous : si on croise une ouverture ou une fermeture, on réagit comme expliqué plus haut. Si on croise une fin de fichier (eof), on renvoie LexingError : les commentaires multilignes devront obligatoirement être fermés correctement (on évite ainsi les erreurs étranges qui viennent d'un commentaire qu'on croyait qu'il n'était pas fermé mais qu'en fait il n'est fermé). Si on rencontre n'importe quoi d'autre (rappelez-vous de _...), on rappelle notre règle sur la suite du code.

Sans plus tarder, voilà donc le code complet de notre analyseur lexical :

```
{
  open Parser
  exception LexingError
}

let digits = ['0' - '9']
```

```
let alpha = ['a' - 'z' 'A' - 'Z']
let empty = ['\n' '\t' ' ']

rule lexer = parse
  | "//" [^'\n']* '\n'? { lexer lexbuf }
  | "/*" { comment 1 lexbuf }
  | eof { Eof }
  | empty+ { lexer lexbuf }
  | "true" { True }
  | "false" { False }
  | "not" { Not }
  | "and" { And }
  | "or" { Or }
  | "if" { If }
  | "while" { While }
  | "print" { Print }
  | "(" { LeftPar }
  | ")" { RightPar }
  | "{" { LeftCurly }
  | "}" { RightCurly }
  | "+" { Plus }
  | "*" { Times }
  | "-" { Minus }
  | ">" { Greater }
  | "=" { Equal }
  | ":=" { Affect }
  | digits+ as n { Int (int_of_string n) }
  | alpha (alpha | digits)* as v { Var v }

and comment depth = parse
  | "/*" { comment (depth + 1) lexbuf }
  | "*/" {
    if depth = 1 then lexer lexbuf
    else comment (depth - 1) lexbuf
  }
  | eof { raise LexingError }
  | _ { comment depth lexbuf }
```

Vous avez donc la solution complète de la gestion des commentaires. Vous aurez peut-être remarqué la petite subtilité sur les commentaires unilignes : le saut de ligne final est facultatif. C'est pour permettre à l'utilisateur d'écrire un commentaire uniligne en fin de fichier sans avoir à terminer sa ligne finale par un saut de ligne (notez que bon, en vrai il devrait de toute façon le faire, mais ne soyons pas plus royalistes que le roy).

4.3.7 Utilisation pratique de notre analyseur

Félicitations, vous avez maintenant un analyseur lexical complet ! Il ne nous reste plus qu'à voir son utilisation pratique, et nous pourrons passer au gros mais intéressant morceau de l'analyse syntaxique.

La distribution standard d'OCaml fournit un module Lexing, dédié comme son nom l'indique à l'analyse lexicale. Il fournit entre autres quelques fonctions de base permettant de construire des lexers, que vous devriez regarder un jour où vous avez un peu de temps, au moins pour la culture. Il nous intéresse parce qu'il fournit également un type lexbuf, qui doit maintenant vous dire quelque chose : il correspond à un « buffer » (un tampon en français) lexical. Ce type a un comportement ressemblant aux flux (*stream* en anglais) pour ceux qui connaissent : il s'agit pour simplifier d'une liste dont la tête n'est calculée que quand on la demande explicitement (on dit que la liste est **paresseuse**), et est supprimée de la liste une fois qu'on l'a produite (la liste est **destructive**). Rappelez-vous, le mot-clef lexbuf d'OCamllex fonctionnait exactement de cette façon : nos règles le prenaient en argument, renvoyant le lexème correspondant au premier mot rencontré, et ce mot était retiré du « code ». Il s'agit donc d'un flux de caractères.

Notre analyseur lexical produira des fonctions dont le paramètre aura ce type (en plus des éventuels paramètres supplémentaires de nos règles). Il faudra donc, à partir de notre code (qui sera contenu soit dans un fichier, soit dans une chaîne de caractères), produire une valeur de type Lexing.lexbuf. Vous vous en doutez, le module Lexing propose plusieurs fonctions pour faire ça : vous avez par exemple Lexing.from_string pour convertir une valeur de type string, ou Lexing.from_channel pour convertir une valeur de type in_channel.

Pour qu'il génère ces fonctions, exécutez la commande suivante :

```
$ ocamllex lexer.mll
```

Ocamllex créera alors un fichier lexer.ml (vous pouvez changer le nom à l'aide de l'option -o), qui contiendra les fonctions lexer et comment. Lexer.lexer sera donc de type Lexing.lexbuf -> Tokens.token, ce que vous aurez compris tout seul si vous avez bien suivi.

En bonus, voilà un petit programme qui vous permettra de tester votre analyseur lexical (appelez-le test_lexer.ml) :

```
open Tokens

let string_of_token = function
  | True -> "True"
  | False -> "False"
  | Not -> "Not"
  | And -> "And"
  | Or -> "Or"
  | Greater -> "Greater"
  | Equal -> "Equal"
  | LeftPar -> "LeftPar"
  | RightPar -> "RightPar"
  | LeftCurly -> "LeftCurly"
  | RightCurly -> "RightCurly"
  | Affect -> "Affect"
  | If -> "If"
```

```
  | While -> "While"
  | Print -> "Print"
  | Plus -> "Plus"
  | Times -> "Times"
  | Minus -> "Minus"
  | Int n -> "Int " ^ string_of_int n
  | Var v -> "Var " ^ v
  | Eof -> "Eof"

let lexbuf = Lexing.from_channel (open_in Sys.argv.(1))

let rec print_code lexbuf =
  let t = Lexer.lexer lexbuf in
    print_endline (string_of_token t);
    if t <> Eof then print_code lexbuf

let () =  print_code lexbuf
```

Compilez ce code avec :

```
$ ocamlbuild test_lexer.native
```

Si votre analyseur lexical a été correctement écrit, vous devriez avoir un message sans erreur. Vous pouvez ensuite le tester avec :

```
$ ./test_lexer.native test.imp
Var n
Affect
Int 5
Var f
Affect
Int 1
While
Var n
Greater
Int 0
LeftCurly
Var f
Affect
Var f
Times
Var n
Var n
Affect
Var n
Minus
```

```
Int 1
RightCurly
Print
Var f
Print
Int 3
Plus
Int 4
Times
Int 5
Eof
```

Ce résultat ayant été obtenu à partir du fichier test.imp suivant :

```
// On initialise les variables
n := 5
f := 1

/*
  boucle principale
  /* commentaire imbriqué */
*/

while n > 0 {
    f := f * n
    n := n - 1
    }


print f
/* On peut même print-er des expressions numériques */
print 3 + 4 * 5
```

Si vous n'avez pas ce résultat, il y a un problème chez vous. Si vous ne trouvez pas, vous avez tout le corrigé du code, profitez-en maintenant pour regarder ce qui ne va pas et régler ce problème.

Et voilà, nous en avons terminé avec l'analyse lexicale ! Maintenant que vous n'avez plus de problème avec cette partie, nous allons pouvoir passer à l'analyse syntaxique. N'hésitez pas, comme je l'ai dit plusieurs fois, à aller regarder plus en détail la théorie des expressions rationnelles et des automates finis : c'est relativement simple, et c'est toujours intéressant (et vous comprendrez probablement mieux ce qui va suivre).

5 L'analyse syntaxique

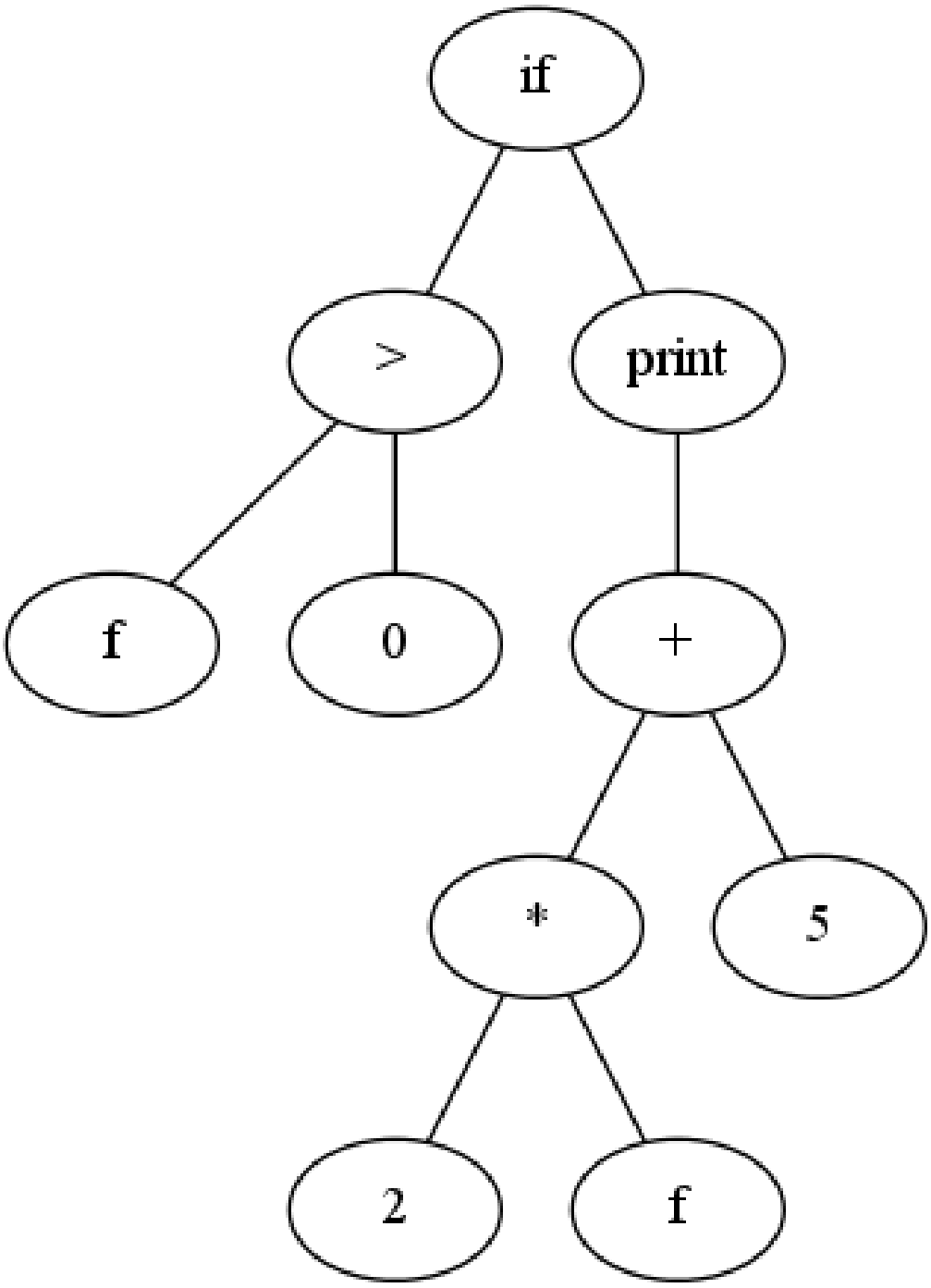
5.1 Arbres syntaxiques

Lors de la première étape, l'analyse lexicale, nous avons transformé le code source initial en une suite de lexèmes connus de notre langage. Si vous vous souvenez bien, nous allons maintenant lier ces lexèmes entre eux : nous allons former des phrases avec les mots que nous avons obtenu.

Prenons un exemple simple avec le code suivant :

```
1 if f > 0 {
2     print 2 * f + 5
3 }
```

En oubliant pour l'instant les lexèmes, analysons donc les rapports entre les différents mots. Intéressons-nous d'abord à l'intérieur de la condition. Nous avons un print, suivi d'une opération arithmétique. print étant un mot-clef connu de notre langage, nous savons que cette ligne sera un appel à notre fonction print, avec comme paramètre 2 * f + 5. Notons-le de cette façon : Print ("2 * f + 5"). Il nous faut maintenant analyser ce paramètre pour compléter le découpage de la ligne. Il s'agit d'une opération à deux opérateurs, * et +. Nous ne pouvons donc a priori pas faire comme print, à savoir décomposer directement en une opération et son ou ses opérandes. Sauf que... nous avons les règles de priorité ! Ces règles nous disent qu'on commence par calculer 2 * f, puis qu'on ajoute 5. Donc nous avons bien une opération, +, dont les opérandes sont 2 * 5 et f. Autrement dit, quand on a une opération faisant intervenir des * et des +, on commence par analyser l'expression * puis on prend son résultat comme opérande de +. Le résultat final de cette ligne est donc : Print(Add(Mul(2, "f"), 5)). On fait pareil avec le if : on peut dire que If a deux arguments, "f > 0" et la ligne que nous venons d'analyser, ce qui nous donne finalement une décomposition qui, au lieu de s'écrire linéairement, se représente plus intuitivement comme l'arbre suivant :



Comme vous pouvez le constater, c'est un arbre *n-aire* (le nombre de fils de chaque noeud n'est pas fixé - en pratique il ne dépassera pas 2 dans notre cas), où chaque « mot » de notre langage représente un

noeud (ou une feuille). Il montre bien la relation entre nos mots : la portée de chacun, ses paramètres… Cet arbre particulier est appelé **arbre syntaxique abstrait**. On abrège très souvent tout le temps en AST (*abstract syntax tree*). Et comme vous l'avez remarqué, à chaque noeud ou feuille de notre arbre, on trouve un mot correspondant exactement à un lexème du langage : on a donc complètement décomposé notre liste de lexèmes dans notre AST.

Afin de vous assurez que vous maîtrisez le concept d'arbre syntaxique, vous avez ci-dessous l'arbre syntaxique correspondant au code de test de notre analyseur lexical (redonné pour mémoire). Assurez-vous de bien comprendre comment il est construit (ce n'est pas très compliqué, mais c'est important). Notez que Seq désigne la suite de deux instructions.

Voici donc la source :

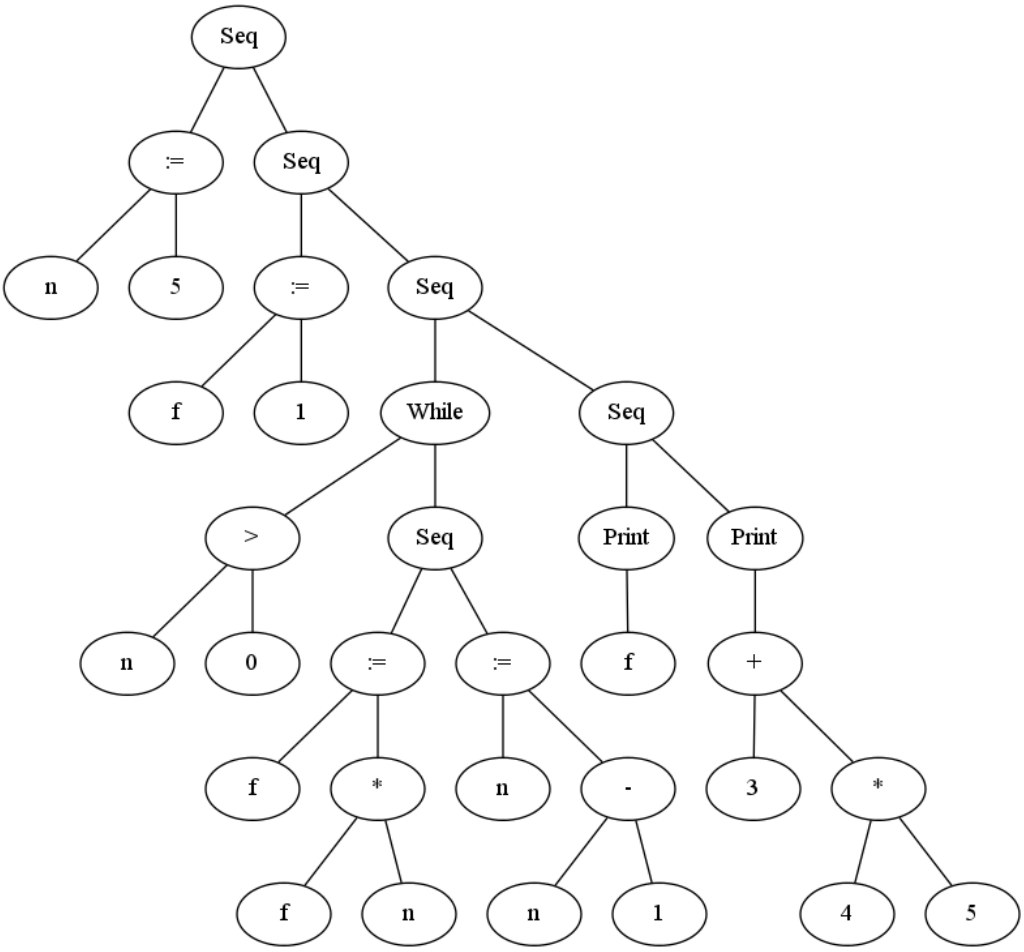
```
// On initialise les variables
n := 5
f := 1

/*
  boucle principale
  /* commentaire imbriqué */
*/

while n > 0 {
    f := f * n
    n := n - 1
    }


print f
/* On peut même print-er des expressions numériques */
print 3 + 4 * 5
```

Et l'arbre qui en découle :



Vous pouvez remarquer que le constructeur Seq prend deux sous-AST en paramètre. On aurait aussi pu lui en faire prendre un nombre quelconque (donc une liste d'AST) : les deux solutions sont en fait à peu près équivalentes, celle que nous avons choisi revenant en fait à construire des listes avec Seq au lieu de Cons (ou : :, comme vous voulez).

Il ne nous reste plus qu'à définir notre AST dans un fichier ast.ml :

```
type command =
  | Affect of string * aexp
  | Print of aexp
  | If of bexp * command
  | While of bexp * command
  | Seq of command * command

and aexp =
  | Int of int
```

```
10      | Var of string
11      | Minus of aexp * aexp
12      | Neg of aexp
13      | Plus of aexp * aexp
14      | Times of aexp * aexp
15
16
17  and bexp =
18      | And of bexp * bexp
19      | Equal of aexp * aexp
20      | False
21      | Greater of aexp * aexp
22      | Not of bexp
23      | Or of bexp * bexp
24      | True
```

Vous aurez remarqué qu'il est composé de 3 différents types : nous reviendrons dessus plus tard. Ils sont a priori assez explicites pour que vous n'ayiez pas besoin de plus d'explications.

5.2 Les grammaires hors-contexte

Maintenant que nous connaissons la forme de la sortie de notre analyseur syntaxique, il nous faut transformer notre liste de lexèmes en AST. Pour faire simple, commençons par les expressions arithmétiques. Nous allons les redéfinir plus formellement, de la façon suivante :

1. Les entiers sont des expressions arithmétiques ;
2. Les variables sont des expressions arithmétiques ;
3. Si A et B sont des expressions arithmétiques, alors A + B, A * B et A - B sont des expressions arithmétiques, avec les priorités usuelles ;
4. Si A est une expressions arithmétique, alors - A et (A) sont des expressions arithmétiques.

Vous aurez remarqué que je note + pour avoir des formules plus lisibles, mais en réalité il s'agit bien du lexème Plus.

Pour l'analyse lexicale, nous avons utilisé les expressions rationnelles. Comme il s'agit toujours de reconnaître certains motifs, nous pourrions être tentés de recommencer, en remplaçant les caractères par des lexèmes. Seulement, ça ne marchera pas. En effet, le langage des expressions arithmétiques est *non-régulier* : on ne peut pas le décrire à l'aide d'une expression régulière. En fait, on ne peut même pas écrire d'expression régulière qui décrive un mot composé de n parenthèses ouvrantes suivi de n parenthèses fermantes. Une explication « avec les mains » à cette contrainte est liée aux automates finis, qui, rappelez-vous, sont les objets permettant de faire « fonctionner » les expressions régulières. Un automate fini est composé d'un certain nombre N d'états, et il ne peut pas « retenir » plus de N caractères. Or ici, on veut pouvoir retenir un nombre arbitrairement grand de parenthèses gauches pour savoir si on a bien le bon nombre de parenthèses droites pour les fermer : il faudrait donc un automate *infini*, qui n'est donc associé à aucune expression rationnelle.

Il nous faut donc trouver une autre forme de description de notre langage. Elle découle en fait directement de notre définition précédente : comme vous l'avez constaté, cette définition est récursive, et c'est précisément cette récursivité qui va nous permettre de nous en sortir. Écrivons donc :

```
A:
  | Int
  | Var
  | A + A
  | A - A
  | A * A
  | - A
  | (A)
```

Cette écriture, qui n'est autre que notre définition avec une syntaxe un peu plus formelle (A désignant les expressions arithmétiques), est celle des **grammaires hors-contexte**. A est appelé **terme** de la grammaire, et les différents « cas » (Int, A + A...) des **productions**.

On peut comparer les grammaires hors-contexte aux expressions rationnelles, mais en plus expressif : on peut décrire (strictement) plus de choses à l'aide d'une grammaire hors-contexte qu'en utilisant des expressions rationnelles. D'ailleurs, nous allons voir que les grammaires hors-contexte « incluent » les expressions rationnelles. Pour avoir ces dernières sur un alphabet (tout à l'heure les caractères des String, ici les lexèmes), il faut avoir, en plus de ces caractères et de ε qui désigne le mot vide, 3 choses : la concaténation, l'union | et l'étoile *. Tout le reste n'est que du « sucre » : par exemple, s+ peut s'écrire comme ss*, et s ? peut s'écrire comme s| .

Sans reconstruire rigoureusement les expressions rationnelles, voyons comment nous pouvons écrire un terme qui décrit la même chose que (a|b)*ba :

```
A:
  | a

B:
  | b

AorB:
  | A
  | B

AorBstar:
  |
  | AorB AorBstar

Final:
  | AorBstar B A
```

Comme vous pouvez le voir, nous avons un terme (Final) qui décrit la même chose que notre expression rationnelle initiale. Ce qui est intéressant ici, c'est de voir comment nous avons redéfini l'étoile et le tube. Les grammaires hors-contexte permettent donc de faire plus de choses que les expressions rationnelles. On pourrait donc les utiliser aussi pour l'analyse lexicale... Sauf que, comme vous pouvez le voir sur l'exemple précédent, les expressions rationnelles permettent en général une écriture beaucoup plus concise (et l'implémentation derrière est moins lourde).

Nous allons nous arrêter ici pour la théorie et passer à l'utilisation pratique de ce nouvel outil. Comme pour les expressions rationnelles, j'invite le lecteur curieux à se renseigner plus en détail sur les grammaires hors-contexte : il reste pas mal de choses à dire dessus. Vous apprendrez ainsi la signification de termes comme *LL(*)* ou *LR(1)*, vous découvrirez l'analyse descendante et les automates à pile. En poussant un peu plus loin vous entendrez parler de classification des langages, et de tout un tas de choses intéressantes qui vous permettront de briller dans les dîners de la haute société.

5.3 Menhir

5.3.1 Présentation

L'outil que nous allons utiliser pour l'analyse syntaxique s'appelle Menhir. Tout comme Ocamllex pour l'analyse lexicale, il s'agit d'un *générateur d'analyseur syntaxique* : à partir de la description des termes d'une grammaire, il génère un fichier OCaml qui prendra en entrée un flot de lexèmes (sortant d'Ocamllex) et produira en sortie notre AST.

Il s'agit d'une adaptation pour OCaml d'un outil conçu à l'origine pour le langage C, appelé Bison (qui est lui-même une alternative libre au logiciel Yacc). Il existe également un outil appelé Ocamlyacc dont vous entendrez certainement parler : ils se ressemblent beaucoup, mais Menhir est strictement meilleur, c'est pourquoi je vous conseille de l'utiliser dès que vous en avez la possibilité. Pour la petite histoire, Ocamlyacc est l'outil utilisé pour l'analyse syntaxique du langage OCaml (tout comme Ocamllex est utilisé pour son analyse lexicale), car il a l'avantage dans ce cas précis de simplifier le *bootstrap* du langage, étant écrit en C (et non pas en OCaml comme Menhir).

Cette partie vous apprendra ce qu'il faut de Menhir pour interpréter notre petit langage, mais il restera encore beaucoup de choses à dire dessus. Je vous encourage à en lire la documentation pour en savoir plus (par exemple sur le traitement des erreurs et sur la librairie standard, que nous n'abordons pas du tout ici).

5.3.2 Installation

Menhir n'est pas fourni de base avec l'installation d'OCaml, il faudra donc l'installer explicitement vous-mêmes.

Si vous êtes sous Windows, je ne sais pas comment l'installer, mais je suis à peu près persuadé que c'est possible (si quelqu'un l'a fait et peut écrire quelques lignes expliquant la manipulation, je serais ravi de les inclure ici).

Plusieurs distributions Linux l'incluent dans leurs paquets. Pour les utilisateurs d'Archlinux, il existe un paquet `ocaml-menhir` dans l'AUR. Pour les autres distributions, je vous laisse chercher par vous-mêmes, ça ne devrait pas être très compliqué.

Si vous êtes sur un unixoïde ne comprenant pas Menhir dans ses paquets, ou que vous préférez installer vos logiciels à la main, rendez-vous à cette adresse. Téléchargez les sources de Menhir, décompressez-les (`tar -xvf le_fichier.tar.gz`), placez-vous dans le répertoire qui les contient et lancez les commandes (vous avez besoin de l'outil `Make`, qui est de toute façon un indispensable que vous avez certainement déjà) :

```
$ make
$ make install
```

Si tout s'est passé sans erreur, vous disposez maintenant de Menhir sur votre machine. Vous pouvez le vérifier en observant le résultat de la commande suivante :

```
$ menhir
Usage: menhir <options> <filenames>
```

5.3.3 Redéfinissons nos lexèmes

Notre analyseur lexical prendra en entrée un flux de lexèmes, il faut donc qu'il les connaisse d'une façon ou d'une autre. Pour cela, nous allons écrire les lignes suivantes, dans un fichier que vous appellerez parser.mly, afin de tous les redéfinir :

```
%token True False Not And Or
%token Greater Equal
%token LeftPar RightPar LeftCurly RightCurly
%token Affect
%token If While Print
%token Plus Times Minus
%token <int> Int
%token <string> Var
%token Eof
```

Chaque ligne commence par %token, suivi d'un ou plusieurs lexèmes (nous les avons ici regroupés dans un ordre plus ou moins « thématique »). Certains lexèmes peuvent prendre un paramètre : c'est le cas pour nous de Int et Var. Le type de ce paramètre est alors spécifié avant le token, entre chevrons : $< \ldots >$.

Notez qu'une fois que nous aurons terminé l'écriture de notre analyseur syntaxique, nous n'aurons plus besoin du fichier tokens.ml : le type qu'il définit sera contenu dans le code OCaml produit par Menhir.

Après ces lignes, ajoutez un séparateur %% sur une ligne. Il servira à différencier le *header* du fichier (le terme anglais est systématiquement utilisé) des termes et productions les utilisant, que nous placerons après.

5.3.4 Passons aux choses sérieuses

Commençons par écrire les productions des expressions arithmétiques. Notre terme s'appellera aexp. Nous commençons par sa définition structurelle :

```
aexp:
| Int
| Var
| aexp Plus aexp
| aexp Minus aexp
| aexp Times aexp
| Minus aexp
| LeftPar aexp RightPar
```

Le terme est maintenant défini, mais tel quel, il ne nous servira pas beaucoup pour la création de l'AST. Il faut pouvoir récupérer les informations qu'il contient et en faire quelque chose.

Pour cela, chaque production sera suivie, entre accolades {...}, d'un code OCaml qui construira l'AST correspondant. Pour récupérer la valeur associée à un élément de production, vous pouvez le nommer, comme le montre la ligne suivante :

```
aexp:
| x = Int { Ast.Int x }
```

Rappelez-vous que nous avons défini le token Int comme prenant un paramètre de type int. x désigne ce paramètre dans la ligne que nous venons d'écrire. Pour les éléments étant eux-mêmes des termes de la grammaire, la valeur ainsi capturée correspond à la valeur renvoyée par l'analyseur syntaxique lorsqu'il analyse la phrase correspondante. C'est donc la valeur que vous écrivez entre accolades. On peut donc se servir de ces valeurs pour construire récursivement notre ast :

```
aexp:
| x = Int { Ast.Int x }
| v = Var { Ast.Var v }
| x = aexp Plus y = aexp { Ast.Plus (x, y) }
| x = aexp Minus y = aexp { Ast.Minus (x, y) }
| x = aexp Times y = aexp { Ast.Times (x, y) }
| Minus x = aexp  { Ast.Neg x }
| LeftPar x = aexp RightPar { x }
```

Sauf que, comme vous l'aurez évidemment remarqué, il y a un problème : notre parseur (il paraît que le terme existe en français) ne connait pas la priorité de Times sur Plus. En fait, il ne connait même pas la « priorité » de Plus sur Plus, c'est-à-dire l'associativité : comment réduire $1 + 2 + 3$, la racine doit-elle être le premier $+$ ou le deuxième ?

Par défaut, Menhir ne soulèvera pas d'erreur, mais seulement un warning : il appliquera des règles conventionnelles (du style de celles de notre lexeur). Seulement, si tout à l'heure c'était la façon canonique de fonctionner, la règle pour l'analyse syntaxique est de ne pas laisser passer ce genre de conflits (appelé « severe conflict ») où le parseur doit décider tout seul. Pour cela, nous allons déclarer dans le *header* l'associativité et la priorité de nos différents opérateurs. Pour cela, nous utiliserons les mots-clefs %left, %right et %nonassoc : ils définissent chacun l'associativité du lexème qu'ils précèdent, et un lexème écrit après un autre est considéré comme prioritaire sur le premier.

Écrivons donc :

```
%left Plus Minus
%left Times
%nonassoc neg

%%
```

À quoi sert le %nonassoc, puisqu'il n'apporte aucune information supplémentaire d'associativité ? Ici, il sert à définir la priorité de la négation unaire, qui doit être réduite avant les autres opérations. Mais attention, l'opérateur Minus de cette négation est le même que celui de la soustraction : on ne peut donc pas s'en servir dans notre règle. On écrit donc (arbitrairement) %neg pour nommer la règle, et on y fera référence dans la production à l'aide du mot-clef %prec.

Voici donc le code complet à notre stade de l'analyseur syntaxique :

```
%token True False Not And Or
%token Greater Equal
%token LeftPar RightPar LeftCurly RightCurly
%token Affect
%token If While Print
%token Plus Times Minus
%token <int> Int
%token <string> Var
%token Eof

%left Plus Minus
%left Times
%nonassoc neg

%%

aexp:
| x = Int { Ast.Int x }
| v = Var { Ast.Var v }
| x = aexp Plus y = aexp { Ast.Plus (x, y) }
| x = aexp Minus y = aexp { Ast.Minus (x, y) }
| x = aexp Times y = aexp { Ast.Times (x, y) }
| Minus x = aexp  { Ast.Neg x } %prec neg
| LeftPar x = aexp RightPar { x }
```

Les conflits ainsi résolus à l'aide de règles de priorité s'appellent les « benign conflicts ». Menhir ne vous préviendra pas de leur présence (en réalité, il est possible de les éviter complètement, mais ça mènerait à une multiplication assez lourde du nombre de termes qui n'est donc pas intéressante d'un point de vue pratique).

Si vous avez bien compris ce que nous avons fait jusque là, le reste ne devrait vous poser aucun problème. Nous pouvons par exemple enchaîner directement avec les expressions booléennes :

```
%left Or
%left And
%nonassoc Not
%left Plus Minus
%left Times
%nonassoc neg

%%
bexp:
| True { Ast.True }
```

```
| False { Ast.False }
| x = aexp Equal y = aexp { Ast.Equal (x, y) }
| x = aexp Greater y = aexp { Ast.Greater (x, y) }
| a = bexp And b = bexp { Ast.And (a, b) }
| a = bexp Or b = bexp { Ast.Or (a, b) }
| Not b = bexp { Ast.Not b }
| LeftPar b = bexp RightPar { b }
```

5.3.5 Où l'on termine l'écriture de notre AST avec les « commandes » du langage

Jusqu'ici, le découpage de l'AST était assez intuitif (les expressions arithmétiques et booléennes se prêtant naturellement très bien à une transformation en arbre). Pour la suite, voici comment nous allons procéder : il reste les affectations, les boucles, les conditions, les print et les séquences de ces opérations. Tout cela est regroupé dans le type Ast.command. Nous aurons un terme command, qui correspondra à chacune de ces possibilités prise individuellement (donc pas la séquence), une règle sequence qui correspondra à une liste de commandes successives, et une règle prgm qui correspond à une sequence suivie de Eof.

Il est important de retenir cette dernière particularité : vous devez penser à gérer correctement Eof dans votre parseur, sans quoi vous aurez des erreurs de type end-of-stream conflict et, pour faire simple, ça ne marchera pas. Une fois ce découpage fait, le codage de ce qui reste du parseur est relativement aisé, en voici donc la source complète :

```
%{
(* Here can come OCaml code *)
%}

%token True False Not And Or
%token Greater Equal
%token LeftPar RightPar LeftCurly RightCurly
%token Affect
%token If While Print
%token Plus Times Minus
%token <int> Int
%token <string> Var
%token Eof

%start <Ast.command> prgm

%left Or
%left And
%nonassoc Not
%left Plus Minus
%left Times
%nonassoc neg

```

```
%%

prgm:
| s = sequence Eof { s }

sequence:
| c = command { c }
| c = command s = sequence { Ast.Seq (c, s) }

command:
| v = Var Affect x = aexp { Ast.Affect (v, x) }
| Print x = aexp  { Ast.Print x }
| If b = bexp LeftCurly c = sequence RightCurly { Ast.If (b, c) }
| While b = bexp LeftCurly c = sequence RightCurly { Ast.While (b, c)↩
    }

aexp:
| x = Int { Ast.Int x }
| v = Var { Ast.Var v }
| x = aexp Plus y = aexp { Ast.Plus (x, y) }
| x = aexp Minus y = aexp { Ast.Minus (x, y) }
| x = aexp Times y = aexp { Ast.Times (x, y) }
| Minus x = aexp  { Ast.Neg x } %prec neg
| LeftPar x = aexp RightPar { x }

bexp:
| True { Ast.True }
| False { Ast.False }
| x = aexp Equal y = aexp { Ast.Equal (x, y) }
| x = aexp Greater y = aexp { Ast.Greater (x, y) }
| a = bexp And b = bexp { Ast.And (a, b) }
| a = bexp Or b = bexp { Ast.Or (a, b) }
| Not b = bexp { Ast.Not b }
| LeftPar b = bexp RightPar { b }
```

Il reste donc deux petites choses à expliquer, que vous aurez remarquées vous-mêmes si vous êtes attentifs. Tout d'abord, la possibilité d'écrire du code OCaml en tête du parseur. On aurait ainsi pu y écrire open Ast pour éviter d'écrire Ast devant chaque noeud produit - nous ne l'avons pas fait ici pour qu'on distingue bien les noeuds de l'arbre syntaxique des lexèmes, beaucoup ayant le même nom. La deuxième chose est la déclaration %start : tout-à-l'heure, ocamllex générait une fonction pour chaque règle de notre analyseur lexical. Ici, Menhir génèrera une fonction d'analyse syntaxique pour chaque terme déclaré à l'aide de cette commande. Vous aurez noté qu'on doit écrire nous-mêmes le type de sortie de la fonction produite : en l'occurrence, Ast.command (qui est le type auquel appartient le constructeur Seq).

5.3.6 Compilons !

C'est maintenant le moment de vérité : à l'aide de Menhir, vous allez pouvoir compiler le fichier que vous venez d'écrire vers le code OCaml de l'analyseur syntaxique. Exécutez donc les commandes suivantes :

```
$ ocamlc -c ast.ml
$ menhir --infer --explain parser.mly
```

Si tout se passe bien... vous ne verrez rien. Mais si vous listez votre répertoire, 4 nouveaux fichiers seront apparus : ast.cmi et ast.cmo, qui correspondent à la compilation de notre fichier ast.ml (dont Menhir aura besoin à cause du type qu'il définit), et parser.ml et parser.mli. Vous pouvez d'ailleurs regarder l'interface de notre nouveau *parser* :

```
$ cat parser.mli
exception Error

type token =
  | While
  | Var of (string)
  | True
  | Times
  | RightPar
  | RightCurly
  | Print
  | Plus
  | Or
  | Not
  | Minus
  | LeftPar
  | LeftCurly
  | Int of (int)
  | If
  | Greater
  | False
  | Equal
  | Eof
  | And
  | Affect


val prgm: (Lexing.lexbuf -> token) -> Lexing.lexbuf -> (Ast.command)
```

Comme vous pouvez le constater, on exporte le type de nos lexèmes, ainsi que la fonction dont je vous ai parlé tout à l'heure (nous étudierons son type et son utilisation en détail dans la dernière partie), et une exception qui sera utilisée en cas d'erreur de syntaxe sur les codes que nous analyserons.

Comme je vous l'ai annoncé plus haut, vous pouvez maintenant remplacer `open Tokens` par `open Parser` dans votre fichier `lexer.mll` : c'est notre fichier Menhir qui aura désormais la lourde responsabilité de définir les lexèmes de notre langage.

5.3.7 Au secours, ça ne marche pas !

Si vous avez essayé d'écrire votre code Menhir vous-mêmes, je vous en félicite. Par contre vous aurez peut-être des erreurs ou des avertissements que vous ne comprenez pas. Les options que nous avons donné à Menhir servent à cela : `--infer` lui demande de vérifier lui-même la correction de notre typage dans l'analyseur syntaxique (ce qui fait plus d'erreur à la compilation vers OCaml, mais garantit que le fichier `.ml` produit est bien typé), et `--explain` lui demande de détailler les conflits dans un fichier `parser.conflicts`.

Pour pouvoir bien comprendre ce compte-rendu, il vous faudrait des explications plus poussées sur le fonctionnement de l'analyse syntaxique et des automates à pile, qui n'ont pas leur place ici. Je vous encourage donc encore une fois à vous documenter sur le sujet : c'est indispensable pour une bonne maîtrise des outils que nous avons présentés.

Toutefois, en lisant les quelques lignes générées, vous devriez avoir une idée des productions qui posent problème. En raisonnant par vous-mêmes, vous pourrez trouver les cas qui soulèvent un conflit et chercher une manière de les éviter. Ce n'est pas toujours très facile, mais c'est important de bien analyser la grammaire que vous avez définie : souvent, on a de très bonnes idées à ce sujet, avant de se rendre compte qu'elles ne peuvent tout simplement pas être utilisées à cause d'une ambiguité qu'on ne voit pas tout de suite « à l'oeil nu ».

Une fois que tout fonctionne bien, ça y est, vous en avez fini avec l'analyse syntaxique - et avec ça, le plus gros du travail sur notre interpréteur. Encore un dernier petit effort et vous aurez enfin votre petit langage de programmation !