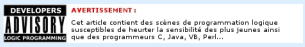
Méthodes de réalisation d'un analyseur syntaxique en Prolog

par Guillaume Piolle (Page perso sur DVP)

Date de publication :

Dernière mise à jour : 13/12/06 (v. 0.31)



Cet article est un cours d'introduction au développement d'analyseurs syntaxiques en langage Prolog (programmation logique), par trois méthodes différentes : les différences de listes (*list differences*), les Arbres Infinis Rationnels (AIR) et les *Definite Clause Grammars (DCG)*.

Ce cours comporte une introduction aux grammaires formelles hors-contexte, qui ne saurait être considérée comme un cours complet sur les grammaires, et qui n'est auto-suffisant que dans le cadre de cet article sur la rédaction d'analyseurs syntaxiques en Prolog.

Ce cours présuppose une bonne connaissance des bases du langage Prolog.

I - Introduction : les analyseurs syntaxiques

Qu'est-ce que l'analyse syntaxique ? L'analyse syntaxique est le procédé par lequel on s'assure qu'un texte est conforme à une grammaire, autrement dit qu'il est bien formé, ou syntaxiquement correct. En français par exemple, l'analyse syntaxique permet de vérifier qu'une phrase a bien au moins un groupe nominal (qui sera le sujet) et un verbe, par exemple. Bien entendu la grammaire française est beaucoup plus complexe, et l'analyseur le serait tout autant. Mais c'est le principe : faire en sorte qu'un déterminant ne se balade pas sans un nom ou un pronom, et d'une manière générale que toutes les règles de la grammaire soient respectées.

L'analyse syntaxique est utilisée dans le traitement des langues naturelles (Traitement du Langage Naturel, TLN, ou en anglais *NLP*, *Natural Language Processing*), comme l'exemple que nous venons de voir, mais aussi et surtout dans le domaine de la compilation. En effet une des étapes de la compilation d'un code source en programme informatique exécutable consiste à s'assurer que le code correspond bien aux spécifications du langage de programmation. En C par exemple, cela permet de s'assurer qu'on ne trouve pas de *else* sans *if*, qu'un *if* a bien une condition et une instruction (ou un bloc d'instructions), etc. On peut également utiliser des outils d'analyse syntaxique, de complexité variable, pour s'assurer qu'une entrée utilisateur ou un fichier de données a une structure correcte.

Il existe des outils (comme yacc ou bison) pour réaliser de manière semi-automatique des analyseurs syntaxiques pour un langage donné. Nous ne les utiliserons pas ici.

Dans ce cours, nous allons voir comment réaliser un analyseur syntaxique en Prolog. Nous allons d'abord nous familiariser avec les grammaires formelles, puis nous verrons trois méthodes différentes pour l'analyse syntaxique en Prolog : les différences de listes, les arbres infinis rationnels, et les *Definite Clause Grammars*. A l'exception de cette dernière méthode, nous nous efforcerons de coder dans un prolog ISO, compatible avec les différents prolog disponibles sur le marché (SWI-Prolog, GNU-Prolog, Prolog IV, SICStus Prolog, Qu-Prolog...). Toutefois les exemples présentés seront illustrés avec SWI-Prolog.

II - Grammaires hors-contexte

II-A - Notion de langage formel

Comme nous l'avons dit, notre problème ici est de vérifier qu'un texte est conforme à certaines spécifications syntaxiques, à un langage formel. Le français ou le C peuvent être considérés comme des langages formels. Il nous faut maintenant un formalisme pour établir nos spécifications et travailler dessus : ce sont les grammaires formelles. Une grammaire est un ensemble de règles, qui déterminent très précisément, de manière claire et complète, le langage qu'elle décrit. Une grammaire détermine un et un seul langage, mais un même langage peut être exprimé de plusieurs manières différentes, avec plusieurs grammaires qui seront formellement équivalentes entre elles.

II-B - Définition formelle des Grammaires hors-contexte

Il existe plusieurs sortes (plusieurs "classes") de langages et de grammaires, et nous n'allons pas toutes les étudier car la théorie des langages est une branche à part entière de l'informatique théorique. Les plus curieux pourront aller s'initier aux travaux de Noam Chomsky, Alan Turing ou Alonzo Church, mais qu'ils ne viennent pas se plaindre si leurs amis commencent à les regarder bizarrement...

Les grammaires qui nous intéressent ici sont les grammaires dites *hors-contexte*, ou *context-free* en anglais. Cela signifie, en très très gros, que l'analyse d'un symbole (ou d'un ensemble de symboles) d'un texte sera la même, quels que soit les symboles qui l'entourent. Si nous reprenons notre exemple de la langue française, ça pourrait vouloir dire par exemple que la définition que nous avons pour un groupe verbal sera la même si le sujet comporte un adjectif ou pas (c'est un exemple stupide). Cette restriction nous limite dans la classe des langages que nous pourrons analyser (certains types de langages nous seront inaccessibles avec des grammaires hors-contexte), mais nous permettra de construire des analyseurs syntaxiques plus efficaces.

Les grammaires permettent de former ou de reconnaître les "mots" d'un langage. Les "mots" sont formés de "symboles". Il faut bien comprendre que le "mot" représente l'intégralité du texte que l'on analyse, par exemple une phrase de français, bien formée, serait représentée par un "mot" du langage formel, et les mots de la phrase seraient les "symboles" du langage. Nous nous efforcerons de conserver ce formalisme tout au long de l'article : le "mot" est la totalité de ce que nous analysons, et ce mot est formé de "symboles" issus d'un "alphabet".

Une grammaire hors-contexte se présente formellement sous la forme d'un quadruplet :

- **L'alphabet terminal** du langage, qui comprend les symboles de base du langage, comme par exemple les opérateurs et les mots réservés pour un langage de programmation ;
- **L'alphabet non terminal**, qui nous permettra de décrire les groupements du langage qui ont un sens en soi, comme par exemple la notion de "groupe nominal" en français ou de "fonction" dans un langage de programmation ;
- Le **symbole de départ**, qui est un symbole du vocabulaire non terminal, et qui représentera le mot du langage dans son ensemble (par exemple en programmation, le "mot" du langage qui correspond au code source du programme en entier) ;
- Les **règles de réécriture**, qui donnent la structure des symboles non terminaux du langage (par exemple en décrivant comment construire un groupe nominal à partir d'un déterminant et d'un nom, ou une boucle while à partir d'un mot réservé, d'une condition et d'un bloc d'instructions).

II-C - Notations et exemple

Nous convenons de représenter la grammaire sous la forme d'une liste de règles de réécriture, avec le symbole non terminal en tête, une flèche, et sa décomposition. la règle suivante :

```
GroupeNominal -> Determinant Nom | Determinant Nom Adjectif | je
```

se lit : "un symbole **GroupeNominal** est composé d'un **Determinant** suivi d'un **Nom**, ou d'un **Determinant** suivi d'un **Nom** suivi d'un **Adjectif**, ou du symbole terminal "je". Ici le pipe (|) représente un "ou" logique, qui nous servira à écrire moins de règles pour un même symbole non terminal, et à augmenter la lisibilité. Dans nos conventions d'écriture, pour être cohérent avec le langage Prolog, nous supposerons que les symboles commençant par des majuscules sont des symboles non terminaux (variables), et que ceux commençant par des minuscules sont des symboles terminaux (constantes). Ici, "je" est un symbole terminal, tous les autres sont non terminaux.

On considérera que la première règle de réécriture concerne le symbole de départ du langage, et que les règles de réécriture d'un même symbole sont regroupées (à la manière des règles d'un même prédicat en Prolog). L'ordre dans lequel les symboles non terminaux apparaissent dans la tête des règles de réécriture nous permettra de juger du niveau d'abstraction correspondant. Par exemple, **GroupeNominal** est d'un niveau d'abstraction plus élevé que **Nom**, donc nous ferons en sorte qu'il apparaisse avant. Ceci n'est cependant pas une règle stricte, elle n'est là que pour améliorer la lisibilité.

Nous allons maintenant nous intéresser à un exemple précis de langage, sur lequel nous allons bâtir nos analyseurs syntaxique. Je vous propose de conserver l'exemple de la langue française. Nous allons travailler sur un langage qui permet de faire des phrases relativement simples, avec un groupe nominal (qui contiendra un déterminant et un nom) et un groupe verbal (qui contiendra un verbe transitif direct, et éventuellement un groupe nominal). Nous allons évidemment travailler avec un vocabulaire très restreint. Notre langage permettra de construire les "mots" suivants (qui correspondent à des phrases en français) :

```
le chien mange
le chien mange le chat
```

Pour des raisons de simplicité, nous évacuons tous les problèmes de ponctuation (majuscule de début de phrase, point final) et ne nous intéressons qu'à la succession des mots. La grammaire formelle qui décrit ce langage pourrait s'écrire ainsi, avec notre formalisme :

```
Phrase          -> GroupeNominal GroupeVerbal
GroupeNominal   -> Determinant Nom
GroupeVerbal    -> Verbe | Verbe GroupeNominal
Determinant     -> le
Nom             -> chat | chien
Verbe           -> mange
```

On notera que les symboles non terminaux **Determinant**, **Nom** et **Verbe** contiennent tous les symboles terminaux du langage, tout en permettant leur classification.

Nous allons complexifier un tout petit peu notre grammaire actuelle, qui décrit un langage qui comporte un nombre fini de mots (la taille des mots étant bornée par 5, en nombre de symboles terminaux). Nous allons introduire des propositions subordonnées : maintenant, un groupe nominal pourra inclure une subordonnée du type "qui mange", ou "qui mange le chat", et ce récursivement. Ainsi, les phrases suivantes sont des mots bien formés du langage :

```
le chien mange le chat qui mange le chien
le chien qui mange le chat qui mange le chien mange le chat qui mange le chat qui mange
```

Evidemment, ça n'est pas du Proust, mais bon... Pour atteindre un niveau de sophistication aussi stupéfiant, nous allons rajouter deux symboles non terminaux (Subordonnee et Relatif, qui désignent une proposition subordonnée et un pronom relatif, respectivement), et le symbole terminal "qui". Notre grammaire devient donc :

```
Phrase          -> GroupeNominal GroupeVerbal
GroupeNominal   -> Determinant Nom | Determinant Nom Subordonnee
GroupeVerbal    -> Verbe | Verbe GroupeNominal
Subordonnee     -> Relatif GroupeVerbal
Relatif         -> qui
Determinant     -> le
Nom             -> chat | chien
Verbe           -> mange
```

Il existe d'autres représentations des grammaires formelles hors-contexte. La plus connue est celle de Backus-Naur (*Backus-Naur Form*, ou *BNF*). Nous ne la détaillerons pas ici, car elle est strictement équivalente à notre notation tout en étant un peu plus lourde, mais nous pouvons donner un équivalent en notation BNF de la grammaire que nous avons choisie :

```
<Phrase>          ::= <GroupeNominal> <GroupeVerbal>
<GroupeNominal>   ::= <Determinant> <Nom | <Determinant> <Nom> <Subordonnee>
<GroupeVerbal>    ::= <Verbe> | <Verbe> <GroupeNominal>
<Subordonnee>     ::= <Relatif> <GroupeVerbal>
<Relatif>         ::= qui
<Determinant>     ::= le
<Nom>             ::= chat | chien
<Verbe>           ::= mange
<Adjectif>        ::= blanc | noir
```

III - Ecrire un analyseur en Prolog

III-A - Structure générale de l'analyseur

Attelons-nous maintenant à écrire un prédicat prolog qui nous permette de déterminer si un texte correspond à une grammaire. Ce sera un prédicat unaire, que nous nommerons audacieusement mais fort à propos "analyse", et qui prendra pour unique argument une liste de chaînes de caractères. Le prédicat se terminera avec succès si la liste correspond à un mot du langage, et il se terminera en échec en un temps fini si ce n'est pas le cas. On considère la liste prise en argument comme complètement instanciée (sans variable libre). L'exemple suivant nous montre comment notre prédicat devrait se comporter une fois terminé :

```
?- analyse([le,chien,mange]).
Yes

?- analyse([le,chien,mange,le,chat]).
Yes

?- analyse([le,lion,est,mort,ce,soir]).
No
```

Que va devoir faire notre prédicat ? Il va devoir vérifier que le texte complet peut s'interpréter comme une réécriture du symbole d'entrée de notre grammaire (ici **Phrase**), en découpant le texte en morceaux qui vont devoir être interprétés comme des réécritures de l'un ou l'autre des symboles de la grammaire. Il est donc probable (mais pas obligatoire comme nous le verrons) que nous ayons à écrire un prédicat pour chacun des symboles non terminaux de notre grammaire.

Vous aurez noté, bande de petits malins, que nous n'avons pas mis de guillemets simples pour délimiter les chaînes de caractères. C'est une facilité qui est acceptée par prolog, à une exception près toutefois : les parenthèses, qui doivent toujours être entre guillemets simples, sous peine d'être interprétées comme des éléments d'arbres par prolog :

```
?- analyse(['(',a,+,b,')']).
No

?- analyse(['(','a','+','b',')']).
No

?- analyse([(,a,+,b,)]).
ERROR: Syntax error: Operator expected
ERROR: analyse([(,
ERROR: ** here **
ERROR: a,+,b,)]).
```

III-B - Restrictions sur la grammaire

Si nous voulons construire un analyseur syntaxique en Prolog sur la base d'une grammaire hors-contexte, il y a deux points auxquels nous devons faire attention : la grammaire ne doit être **ni ambiguë ni récursive à gauche**.

Concernant l'ambiguité, cela signifie qu'il ne faut pas qu'un mot du langage puisse être interprété correctement de deux façons différentes. Par exemple, admettons que l'on traite un langage traitant des conjonctions et disjonctions (non parenthésées) entre propositions logiques. Notre vocabulaire terminal comprendra la proposition "p" et la proposition "q", ainsi que les opérateurs "ou" et "et". Construisons une grammaire naïve :

```
Expression -> Proposition | Expression ou Expression | Expression et Expression
Proposition -> p | q
```

Cette grammaire nous permet de traiter des expressions comme :

```
p
p ou q
q et p ou q
```

Seulement, cette grammaire nous pose problème. En effet, la dernière expression donnée en exemple peut être interprétée de deux manières différentes, à savoir "(q et p) ou q" et "q et (p ou q)", suivant que l'on applique d'abord la seconde ou la troisième version de la première règle de réécriture. (1) Pour rendre cette grammaire non ambiguë, il faudrait introduire une priorité entre opérateurs, et donc séparer l'interprétation du "ou" et celle du "et" dans deux règles de réécriture différentes, pour deux symboles non terminaux différents. La grammaire suivante décrit le même langage de manière non ambiguë, et en tenant compte de la priorité des opérateurs (le symbole de départ est ici **Disjonction**) :

```
Disjonction -> Conjonction | Conjonction ou Disjonction
Conjonction -> Proposition | Proposition et Conjonction
Proposition -> p | q
```

Le second problème potentiel est la récursivité à gauche. Ce qu'on entend par là, c'est le fait que dans une règle de réécriture pour le symbole **E**, on trouve le symbole **E** au début de la chaîne de réécriture, comme dans l'exemple suivant :

```
E -> F | E ou F
F -> p
```

Ceci doit absolument être évité, car ça nous empêcherait d'écrire un analyseur syntaxique qui fonctionne (du moins avec les techniques présentées ici). En effet, il partirait en boucle infinie dans l'interprétation de **E** : pour savoir si le texte s'interprète comme un **E**, il faut savoir s'il s'interprète comme un **E**, et donc s'il s'interprète comme un **E**... En Prolog cela se traduira par un appel récursif sans terminaison. Toute grammaire qui comporte cette spécificité devra donc être transormée en une nouvelle grammaire, équivalente à la première, et qui ne soit pas récursive à gauche. C'est heureusement toujours possible. Dans notre cas, on obtiendrait la grammaire suivante :

```
E -> p | p ou E
```

Il faut être d'autant plus vigilant à la récursivité à gauche, qu'elle peut être masquée, par exemple dans l'exemple suivant :

```
E -> F | ...
F -> E ou F
...
```

Dans cet exemple il y a une récursivité à gauche masquée (indirecte) entre **E** et **F**. Pour savoir si le texte s'interprète comme un **E**, il faut savoir s'il s'interprète comme un **F**, et donc s'il s'interprète comme un **E**, et donc s'il s'interprète comme un **F**... Ce type de cas doit absolument être évité aussi.

Si nous reprenons la grammaire exemple sur laquelle nous avons choisi de travailler, nous pouvons constater avec bonheur et soulagement qu'elle n'est ni ambiguë ni récursive gauche :

```
Phrase         -> GroupeNominal GroupeVerbal
GroupeNominal  -> Determinant Nom | Determinant Nom Subordonnee
GroupeVerbal   -> Verbe | Verbe GroupeNominal
Subordonnee    -> Relatif GroupeVerbal
Relatif        -> qui
Determinant    -> le
Nom            -> chat | chien
Verbe          -> mange
```

III-C - Et pourquoi pas la concaténation ?

Vu les objectifs que nous nous sommes fixés, on pourrait se dire, empreints d'une naïveté innocente : "Je ne vois vraiment pas où est le problème. Pour chaque symbole non terminal, je vais prendre mon texte à analyser, utiliser le prédicat append, qui fait de la concaténation entre listes, pour découper mon texte en autant de morceaux que nécessaire, et dire que chaque morceau va s'interpréter comme ci ou comme ça..." Voici ce que ça donnerait si l'on essayait d'écrire un interpréteur sur ce principe (on ne donne que le prédicat pour le symbole de départ de notre grammaire) :

```
analysePhrase(L) :-
 append(GroupeNominal, GroupeVerbal, L),
```

```
 analyseGroupeNominal(GroupeNominal),
 analyseGroupeVerbal(GroupeVerbal).
```

Ici nous disons bien que pour que L soit une phrase (soit un "mot" de notre langage), il faut qu'il puisse être décomposé en une liste GroupeNominal et une liste GroupeVerbal, qui puissent être interprétées respectivement comme un groupe nominal et un groupe verbal. Si nous suivons ce mode de construction, nous pouvons donner un analyseur syntaxique complet suivant ce modèle :

```
/*
 AnalysePhrase/1
 Détermine si la liste passée en argument est une phrase bien formée,
 conformément à notre grammaire.
*/
analysePhrase(P) :-
 append(GroupeNominal, GroupeVerbal, P),
 analyseGroupeNominal(GroupeNominal),
 analyseGroupeVerbal(GroupeVerbal).

/*
 AnalyseGroupeNominal/1
 Détermine si la liste passée en argument correspond à un symbole
 GroupeNominal dans notre grammaire.
*/
analyseGroupeNominal(GroupeNominal) :-
 append(Determinant, Nom, GroupeNominal),
 analyseDeterminant(Determinant),
 analyseNom(Nom).
analyseGroupeNominal(GroupeNominal) :-
 append(Determinant, Suite, GroupeNominal),
 append(Nom, Subordonnee, Suite),
 analyseDeterminant(Determinant),
 analyseNom(Nom),
 analyseSubordonnee(Subordonnee).

/*
 AnalyseGroupeVerbal/1
 Détermine si la liste passée en argument correspond à un symbole
 GroupeVerbal dans notre grammaire.
*/
analyseGroupeVerbal(GroupeVerbal) :-
 analyseVerbe(GroupeVerbal).
analyseGroupeVerbal(GroupeVerbal) :-
 append(Verbe, GroupeNominal, GroupeVerbal),
 analyseVerbe(Verbe),
 analyseGroupeNominal(GroupeNominal).

/*
 AnalyseSubordonnee/1
 Détermine si la liste passée en argument correspond à un symbole
 Subordonnee dans notre grammaire.
*/
analyseSubordonnee(Subordonnee) :-
 append(Relatif, GroupeVerbal, Subordonnee),
 analyseRelatif(Relatif),
 analyseGroupeVerbal(GroupeVerbal).

/*
 AnalyseRelatif/1
 Détermine si la liste passée en argument est un singleton (symbole
 terminal) correspondant à un symbole Relatif dans notre grammaire.
*/
analyseRelatif([qui]).

/*
 AnalyseDeterminant/1
 Détermine si la liste passée en argument est un singleton (symbole
 terminal) correspondant à un symbole Determinant dans notre grammaire.
*/
analyseDeterminant([le]).
```

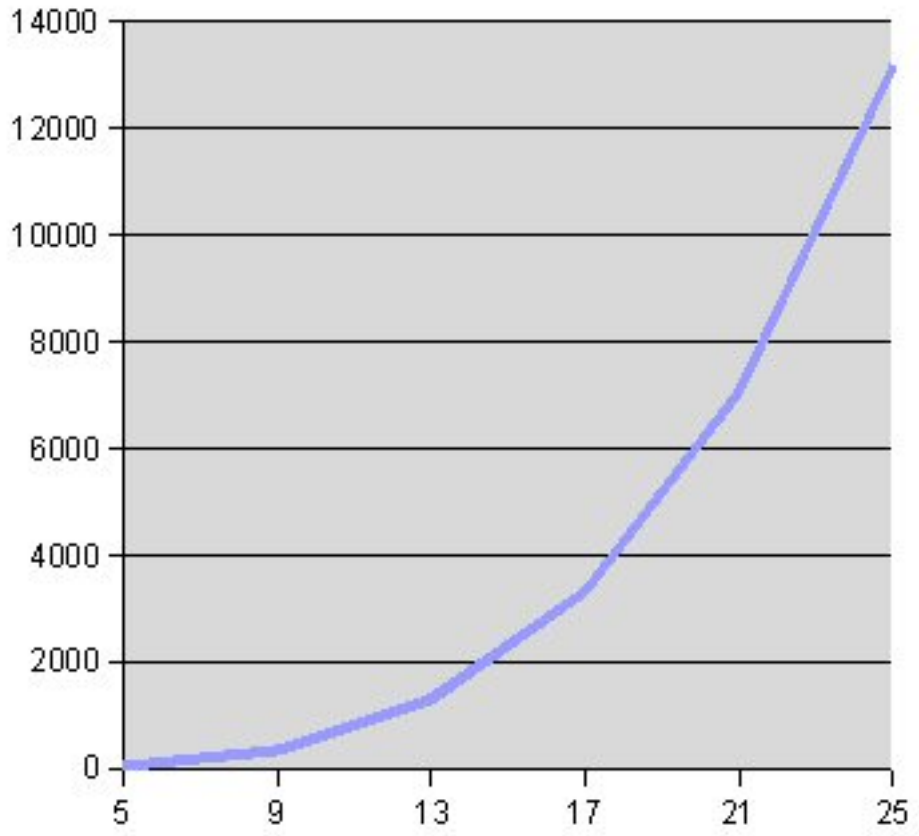
```
/*
 analyseNom/1
 Détermine si la liste passée en argument est un singleton (symbole
 terminal) correspondant à un symbole Nom dans notre grammaire.
*/
analyseNom([chat]).
analyseNom([chien]).

/*
 analyseVerbe/1
 Détermine si la liste passée en argument est un singleton (symbole
 terminal) correspondant à un symbole Verbe dans notre grammaire.
*/
analyseVerbe([mange]).
```

D'accord, j'avoue, c'est un analyseur syntaxique qui fonctionne : Si vous lui donnez une liste en entrée, il pourra vous dire si oui ou non elle correspond à un mot du langage. Cependant, il a deux énormes problèmes, qui vont le rendre inutilisable. Ces deux défauts sont tous les deux liés à l'utilisation de la concaténation (prédicat append/3).

Tout d'abord, notre analyseur s'appuie sur la concaténation et les retours en arrière de l'interpréteur prolog pour rechercher la bonne décomposition de chaque symbole non terminal. En conséquence, à chaque étape prolog étudie toutes les possibilités de suffixes/préfixes, ce qui n'est pas du tout optimal. Du coup, il n'est pas difficile de trouver des types de phrases dont l'interprétation a une complexité au moins polynomiale en fonction de la taille de la liste d'entrée. Le diagramme suivant montre cette complexité (exprimée en nombre d'inférences pour une interprétation donnée, directement lié au temps d'exécution du programme) pour des phrases de la forme suivante :

```
le chien mange le chien
le chien qui mange le chien mange le chien
le chien qui mange le chien qui mange le chien mange le chien
...
```



Complexité de l'analyseur : Nombre d'inférences en fonction de la taille de la liste en entrée

Cette progression rapide du nombre d'opérations à exécuter nous fait entrevoir le hic : il n'est pas raisonnable d'utiliser un tel analyseur dans un compilateur censé traiter des programmes qui peuvent avoir une taille conséquente. Pour dire ça d'une autre manière, vous n'avez pas envie que le temps d'exécution de l'analyseur syntaxique double à chaque fois que vous ajoutez cinq ou six mots à votre texte d'entrée ! Or c'est exactement ce qui se passe ici. Mais ne désespérez pas tout de suite, nous allons vous proposer des solutions plus efficaces, qui auront une complexité bien moindre.

Le second petit problème est un peu plus subtil à appréhender. Il apparaît lorsqu'on passe une variable libre en argument à l'interpréteur. Dans ce cas, le programme donne deux unifications simples, puis part en boucle infinie sur la suivante. En effet nous avons vu que notre grammaire autorisait des mots de taille potentiellement infinie, et la construction de notre analyseur fait que c'est un mot de ce type-là qui est recherché. Les autres solutions que nous allons voir, au contraire, chercheront des mots de taille croissante. De cette manière, ils donneront potentiellement une liste infinie de possibilités (car notre langage comprend une infinité de mots bien formés), mais pour tout nombre entier N, les N premières unifications auront une taille finie (et donc seront exploitable). On n'aura donc pas ce problème de boucle infinie après quelques solutions seulement.

IV - Méthode des différences de listes

IV-A - Introduction aux différences de listes

IV-B - Ecriture de l'analyseur

V - Méthode des Arbres Infinis Rationnels (AIR)

V-A - Introduction aux arbres infinis rationnels

V-B - Représentation de la grammaire par des AIR

V-C - Ecriture d'un anlyseur générique

VI - Méthode des Definite Clause Grammars (DCG)

VI-A - Extensions à Prolog pour les DCG

VI-B - Utiliser les Definite Clause Grammars

VII - Remerciements

VIII - Bibliographie et liens

"Techniques et outils pour la compilation", par Henri Garreta (chapitre 3, analyse syntaxique, pp 19-36)

Page man de yacc

Manuel de yacc

Bison GNU project

1 : Cette grammaire est également récursive gauche, ce qui est le second problème que nous traitons ici.