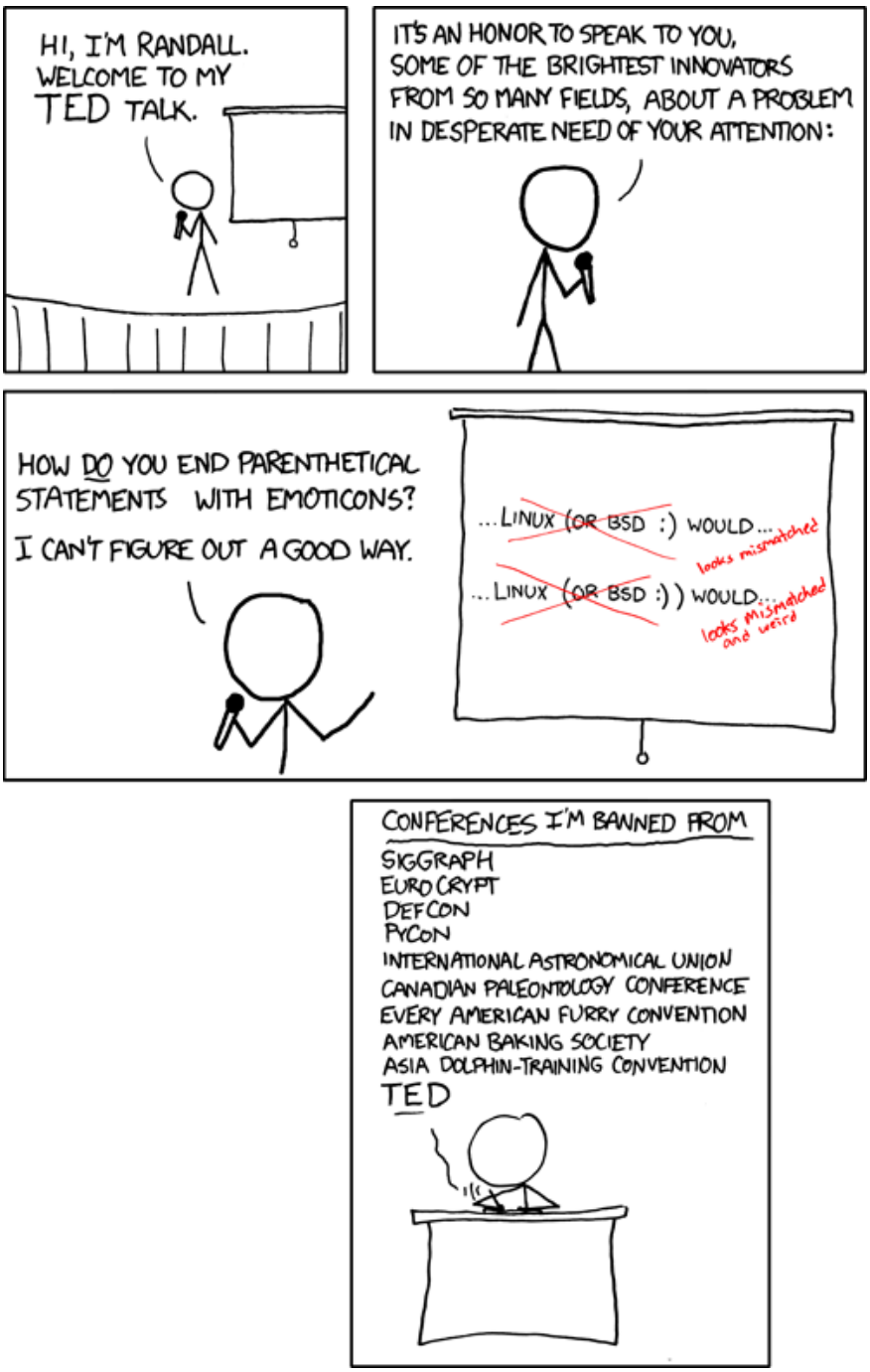
Van De Walle Bernard
Francois Thibault
Van Der Essen Frédéric

INGI2132 : Langages et traducteurs

Rapport intermédiaire 2

Prof. B. Le Charlier



It is now possible with the HAPPY-) programming language!

Introduction

Syntaxe concrètes

Notre langage s'inspire fortement du lisp, langage qui nous attirait par sa simplicité et son élégance. Toutes les structures sont constituées de liste d'élément séparés par des espaces, clos par des parenthèses.

La syntaxe BNF suivante aurait pu être nettement plus simple, cependant nous avons choisi de rendre explicites certains éléments de la syntaxe. Ainsi nous différentions le Nom (¡name¿) de la fonction de ses arguments, même si dans certains cas ce sont des entités syntaxiques identiques.

Les appels de fonction sont de forme *(Nom arg1 arg2 ...)*. A chaque opérateur du langage correspond une fonction, par exemple *(+ 1 2)* Cependant nous avons donné un statut spécial de ¡builtin_call¿ à celles-ci afin d'empècher l'utilisateur de les redéfinir, et l'on espère, de simplifier l'implémentation du futur interpréteur en lui permettant de repérer directement les fonctions internes.

Read et write sont eux aussi implémentés en tant que fonction, la fonction write renvoyant l'argument donné en plus de l'imprimer sur la sortie standard, permettant de les intégrer à des expressions normales.

Cette syntaxe a l'avantage d'avoir peu de caractères réservés $\{(,), ., [,]\}$, et peu de restrictions sur les identifiants (ceux ci ne peuvent commencer par un chiffre). Il est donc facile de créer de nouveaux opérateurs, parfois amusant comme des smileys , ce qui a donné le nom de notre langage.

Notre langage diffère cependant du lisp dans le fait qu'il n'est pas composé d'une seule expression, mais d'une suite de fonctions et méthodes, et que celles-ci sont composées de liste d'instructions. les instructions peuvent ètre des expressions dont la valeur est ignorée. Seul if, while, skip et return ne sont pas des expressions. Le langage ainsi défini est donc impératif, mais garde un petit coté fonctionnel.

Les objets sont implémentés comme des listes de longueur fixe qui peuvent être crées dynamiquement grâce à la fonction new.

Les commentaires sont entre brackets et leur contenu est remplacé par un espace par l'analyseur lexical.

L'ensemble des caractères acceptables par le programme est constitué d'un sous-ensemble de l'ASCII, constitué de l'union de ¡caracter¿ et de ¡reserved_caracter¿. Les caractères Tab et NewLine sont aussi reconnu mais leur valeur peut dépendre de la plateforme.

0.1 Syntaxe initiale

Voici la syntaxe en notation BNF.

```
<Reserved_words>    ::= set | if | while | write | fun | method
```

```
      | null | true | false | read | this
<bin_id> ::=  + | - | * | / | % | '|' | & | < | > | <= | >=
<un_id>  ::=  ! | neg | new
<Reserved_caracter> ::=  ( | ) | . | [ | ]
<digit> ::= 0..9
<pnumber> ::=  <digit> | <number> <digit>
<number>  ::= - <pnumber> | <pnumber>
<caracter> ::= a..z | A..Z | ! | @ | # | $ | % | ^ | & | *
| { | } '|' | | _ | , | ? | - | + | = | /
| \\ | > | < | : | ; | ~ | \ | "
<sp> ::= lambda |   | NewLine | Tab | <sp> <sp>
<esp> ::=   <sp>
<Id> ::= <caracter> | <Id> <caracter> | <Id> <digit>
<meth_or_fun> ::= <methode> | <function>
<program> ::=  <meth_or_fun> | <program> <sp>  <meth_or_fun>
<function> ::= (<sp> fun <sp> ( <sp> <Name> <esp> <arglist> <sp>)
<sp> <instr_list> <sp>)
<arglist> ::= <Id> | <Id> <esp> <arglist>
<methode> ::= (<sp> method . <Number> <sp>
      ( <sp> <Name> <esp> <arglist> <sp> ) <instr_list>  <sp> )
<instr_list> ::= <instr> | ( <sp> <instr_list_np> <sp> )
<instr_list_np> ::= <instr> | <instr> <esp> <instr_list_np>
<instr> ::= <conditional> | <while_block> | <call>
  | ( <sp> return <esp> <expr> <sp> )
  | (<sp> skip <sp>)
<conditional> ::= (<sp> if <esp> <expr> <esp> <instr_list>
  <esp> <instr_list> <sp> )
<while_block ::= ( <sp> while <esp> <expr> <esp> <instr_list> <sp> )
<call> ::= <function_call> | <method_call> | <builtin_call>
<builtin_call> ::= <assignment> | <read_call> | <write_call> | <arithmetic>
<assignment> ::= ( <sp> set <esp> <left_Id> <esp> <expr> <sp> )
<read_call>  ::= ( <sp> read <sp> )
<write_call> ::= ( <sp> write <esp> <expr> )
<arithmetic> ::= <binary> | <unary>
<binary ::= ( <sp> <bin_id> <esp> <expr> <esp> <expr> <sp> )
<unary>  ::=  ( <sp> <un_id> <esp> <expr> <sp> ) | return | skip
<left_Id> ::= <Id> | <Id> . <expr> | this . <expr>
<function_call>  ::= ( <sp> <Id> <esp> <expr_list_np> <sp> )
<expr_list_np> ::= lambda | <expr> | <expr> <esp> <expr_list_np>
<method_call>  ::= ( <sp> <method_id> <esp> <expr_list_np> <sp> )
<method_id>    ::= <Id> . <Id> | super . <Id>
<expr>         ::= <call> | <left_Id> | <number> | null | true | false | this
```

0.2 Syntaxe de l'analyseur Lexical

Notre analyseur lexical nous permet de parser notre programme et d'en ressortir une liste de lexèmes ordonnés concret. La syntaxe qui sera analysée ne sera bien évidemment pas aussi fine que la syntaxe de base définie dans la section précedente (en effet, quel serait l'objectif de ressortir des jetons contenant chacun un et un seul caractère !). Les lexèmes contiendront donc des unités concrètes tel que les identifieurs, les nombes, les nombres négatifs, les parenthèse, etc etc..

Voici donc la syntaxe interpretée par l'analyseur syntaxique :

```
<Reserved_words>    ::= set | if | while | write | fun | method
      | null | true | false | read | new | return | skip
<meth_or_fun> ::= <method> | <function>
<Program> ::=  <meth_or_fun> | <Program>  <meth_or_fun>
<function> ::= (fun ( Id <arglist> ) <instr_list> )
<arglist> ::= Id | Id <arglist>
<methode> ::= (method ( Number ) (Id <arglist> ) <instr_list> )
<instr_list> ::= <instr> | ( <instr_list_np> )
<instr_list_np> ::= <instr> | <instr> <instr_list_np>
<instr> ::= <conditional> | <while_block> | <call>
  | ( return <expr> )
  | ( skip )
<conditional> ::= (if <expr> <instr_list> <instr_list> )
<while_block ::= ( while <expr> <instr_list>)
<call> ::= <function_call> | <method_call> | <builtin_call>
<builtin_call> ::= <assignment> | <read_call> | <write_call>
                | <arithmetic>
<assignment> ::= ( set <left_Id> <expr> )
<read_call>  ::= ( read )
<write_call> ::= ( write <expr> )
<arithmetic> ::= <binary> | <unary>
<binary ::= (<bin_id> <expr> <expr>)
<bin_id> ::=  + | - | * | / | % | '|' | & | < | > | <= | >=
<unary>  ::=  (<un_id> <expr>)
<un_id>  ::=  ! | neg | new
<left_Id> ::= Id | Id . <expr> | this .  <expr>
<user_call>  ::= ( <Id> <expr_list_np> )
<expr_list_np> ::= lambda | <expr> | <expr> <expr_list_np>
<method_call>  ::= ( <method_id> <expr_list_np>)
<method_id> ::= <Id> . <Id> | super . <Id>
<expr> ::= <call> | <left_Id> | <number> | null | true | false | this
```

0.3 Syntaxe WP de l'analyseur grammatical

La Syntaxe de l'analyse grammaticale est de plus haut niveau car elle n'est définie que sur ce qui est renvoyé par l'analyseur syntaxique. Quelques adaptations ont été nécessaires afin de rendre notre syntaxe Weak Priority.

Premièrement toutes les parenthèses se trouvant à l'intérieur d'une rêgle ont du être sorties en tant que nouvelles règles afin de respecter la précédence entre les parenthèses et les espaces.

Pour élíminer les terminaux vides, nous avons dédoublé les règles : une pour le cas vide, et l'autre pour le cas alternatif. Seul le ¡sp¿ nous posait problème (¡sp¿ représente un espace / indentation facultatif) Nous avons décidé de les remplacer par des ¡esp¿ (indentation obligatoire). Remplacer les ¡sp¿ par des ¡esp¿ au niveau de l'analyse est trivial. Il nous aurait beaucoup plu de retirer entièrement les espaces, mais ceux-ci sont important pour délimiter les points. Il aurait été possible d'insérer des délimiteurs fictifs autour des règles avec point, mais cela semblait moins facile à implémenter au niveau de l'analyseur syntaxique.

Comme la grande partie des règles sont contenues par des parenthèses il n'y a pas eu beaucoup de problème de suffixes ou d'indéterminisme. Les rares problèmes de suffixes ont pu ètre résolus en remontant les règles problématiques à l'intérieur de règles avec parenthèses.

Voici donc la syntaxe WP.

```
<number>::=  <number> | pnumber
<meth_or_fun>  ::=  <methode> | <function>
<Program>  ::=  <meth_or_fun> | <Program> esp <meth_or_fun>

<function>  ::= ( esp fun esp <arglist> esp <instr_list> esp )
<methode>  ::= ( esp method . pnumber esp <arglist> esp <instr_list> esp )

<arglist>::= ( esp <arglist_np> esp )
<arglist_np>  ::=  id | id esp <arglist_np>

<instr_list>  ::=  <instr> | ( esp <instr_list_np> esp )
<instr_list_np> ::=  <instr> | <instr> <instr_list_np>

<instr>  ::=  <conditional> | <while_block> | <call>
           | ( esp return esp <expr> esp)
           | ( esp skip esp )
<conditional>  ::= ( esp if esp <expr> esp <instr_list>esp<instr_list> esp )
<while_block>  ::= ( esp while esp <expr> esp <instr_list> esp )
<call>  ::=  <user_call> | <method_call> | <builtin_call>
<builtin_call>  ::=  <assignment> | <read_call> | <write_call> | <arithmetic>

<assignment>  ::= ( esp set esp id esp <expr> esp )
```

```
              | ( esp set esp id . <expr> esp <expr> )
              | ( esp set esp this . <expr> esp <expr> esp )
<read_call>   ::= ( esp read esp )
<write_call>  ::= ( esp write esp <expr> esp )
<arithmetic>  ::=  <binary> | <unary>
<binary>  ::= ( esp bin_id esp<expr> esp <expr> esp  )
<unary>   ::= ( esp un_id esp <expr> esp )
<user_call>   ::= ( esp id esp  <expr_list_np> esp  ) | ( esp id esp )

<method_call>   ::= ( esp id.id esp <expr_list_np> esp )
                 | ( esp super.id esp <expr_list_np> esp )

<expr_list_np>  ::=  <expr> | <expr> esp <expr_list_np>
<expr>  ::=  <call>  | <number> | null | true | false | this | id | id.<expr>
           | this.<expr>
```

1 Exemple de programmation HAPPY

1.1 Exemple1

Un programme complet.

```
(fun (:-D) (write 42))
(fun (** a b) (
  (set cpt b)
  (set res 0)
  (while (> cpt 0)
    (
      (set r (* res b))
      (set cpt (- cpt 1))
    )
  )
  (return R)
))
(fun (main) (
  (HelloWorld)
  (set B (** 4 2))
  (return (write B))
))

(method (4) (-> level) (return this.level))
```

1.2 Exemple2

Exemple de code intéressant.

```
(method.2 (next) (return this.1))
(method.2 (val)  (return this.2))
  [Creation d'un noeud de linked list]
(fun (N x) ((set n (new 2)) (set n.2 x) (set n.1 null) (return n)) )
  [Ajoute un noeud a une linked list]
(fun (-> Node NextNode) ( (set Node.1 NextNode) (return Node)) )

(fun (main) (
  (set A (->(N 1) (->(N 2) (->(N 3) (N 4)))))
  (while A (
    (write (A.val))
    (set A (A.next))
  ))
  (return 0)
))
```

1.3 Exemple3

Voici un exemple de programmation Happy :

```
(fun (Adder A B)(
  (set C (+ A B))
  (return C)
)

(fun (exposant A B)(
  (set result A)
  (set I 0)
  (while (I<B)
    (set result (* result A))
    (set I (+ I 1))
  )
  (return result)
  )
)

(method(3) (Moore A B)(
  (return (A*B*1.5))
  )
```

```
)

(fun (main) (
  (set A 5)
  (set B 10)
  (set C (Adder A B))
  (set D (Exposant C B))
  (write (moore A B))
  )
)
```

Dans cet exemple on montre différente manière de traiter des nombres grâce à des opérations relativement simple, tel que l'addition et l'exposant. Cet exemple permet d'illustrer la syntaxe de manière concrète.

2 L'analyseur Lexical

L'analyseur Lexical a pour fonction de sortir une suite de jeton concrêt et de assez haut niveau, permettant de simplifier la tache du vérificateur de grammaire. Plus concrètement, il s'agit dans un premier temps de transformer un fichier texte en une suite de caractère, et de les parser les un après les autres. Chaque caractère donnera lieu à une action précise, par exemple les caractères spéciaux tel que les paranthèse et les points, etc... donnent lieux à un jeton entier. Les caractères qui ne sont pas des caractères spéciaux vont quand à eux remplir un buffer , qui sera rempli des qu'un caractère spécial ou délimiteur sera rencontré. Une fois qu'un caractère de ce type est rencontré, on procédera à l'analyse de ce que contient le buffer, et selon le cas on en déterminera le type du jeton à donner.

La sortie de l'analyseur lexical sera donc une linkedlist contenant les jetons dans l'ordre correspondant au programme.

Voici la spécification de la méthode principale qui permet d'obtenir la linkedlist de jetons :

```
/**
 * @pre Le parser est initialisé avec le programme.
 * @post Le programme est transformé en un tableaux de caractère,
et est analysé caractère par caractère, permettant de créer
une suite logique de jetons.
 * @return La linkedList contenant tout les tokens.
 * @throws LexicalError
 */
```

3 Test Analyseur Lexical

Voici quelques tests qui permettent d'illustrer le comportement de l'-analyseur lexical .Afin de visualiser la sortie de l'analyseur , on sort le type et le nom du jeton à la sortie standard (stdin).

3.0.1 Test qui marche

Voici le programme que l'analyseur lexical doit prendre en entrée pour ce premier test :

```
(fun (test A B C)((return (+ A(+ B C)))))
```

Voici ce qu'on obtient en sortie :

```
token 0 : ( (
token 1 :   esp
token 2 : fun fun
token 3 :   esp
token 4 : ( (
token 5 :   esp
token 6 : test id
token 7 :   esp
token 8 : A id
token 9 :   esp
token 10 : B id
token 11 :   esp
token 12 : C id
token 13 :   esp
token 14 : ) )
token 15 :   esp
token 16 : ( (
token 17 :   esp
token 18 : ( (
token 19 :   esp
token 20 : return return
token 21 :   esp
token 22 : ( (
token 23 :   esp
token 24 : + binary
token 25 :   esp
token 26 : A id
token 27 :   esp
token 28 : ( (
token 29 :   esp
```

```
token 30 : + binary
token 31 :   esp
token 32 : B id
token 33 :   esp
token 34 : C id
token 35 :   esp
token 36 : ) )
token 37 :   esp
token 38 : ) )
token 39 :   esp
token 40 : ) )
token 41 :   esp
token 42 : ) )
token 43 :   esp
token 44 : ) )
token 45 :   esp
```

L'analyseur lexical marche donc parfaitement , et permet de reconnaitre les différents type de token, ainsio que leurs valeurs.

3.0.2 Test qui ne marche pas

Voici un test qui ne marche pas :
(fun (azertyμ A B)((return 3))) Exception in thread "main" list.all.LexicalError : μ

Ce Test ne marche pas car un caractère illégal a été utilisé, et en conséquence l'analyseur lexical le reconnait et l'interdit.

3.1 Autre Test qui ne marche pas

Voici un dernier test afin d'illustrer notre analyseur lexical :

```
.(fun (test A B)((return 3) ))
Exception in thread "main" list.all.LexicalError: Erreur : No starting point
```

Ce test illustre que Un programme ne peut pas démarrer par un point, et cela est bel et bien détecté et catché par notre analyseur lexical.

4 Vérificateur de grammaire

4.1 BNF Parser

Le point de départ du vérificateur de grammaire est le parser BNF. Ce parser permet d'avoir sous une forme utilisable que nous détaillerons plus loin, les règles contenue dans le fichier. Le loader charge le fichier et le lit ligne par ligne.

Loader

```
/**
 * initialize the rules loader, the file should be a valid BNF File
 * that mean every rules look like that
 * nonTerminal ::= rules1 | rules2 | .. | rulesN
 * nonTerminal should be written like that : <id>
 * Terminal should be written like that : 'term'
 * special caracter  ' \ could be in a terminal symbole precede by a \
 *
 * exemple :
 * <E> ::= <T>
 * <E> ::= '\' <T>
 * <E> ::= <T> '+' <E>
 * <T> ::= <F>
 * <T> ::= <F> '*' <T>
 * <F> ::= '\''
 *
 * @param file the path of the file
 * @throws FileNotFoundException
 * @throws IOException
 *
 * the rules can be extract with the method getRules()
 */
```

Avec chaque ligne le loader instancie une rule qu'il place dans une liste. La classe rule parse la ligne et sépare d'un coté le nom et de l'autre une orListe qui représente une liste de règles *ou*. Chaque éléments de la orList est une catList c'est à dire une liste de *Term* qui forme une règles.

4.2 Architecture

Le vérificateur de grammaire est implémenté comme une série de test statique correspondant aux conditions de Weak Priority.

Tous ces tests prennent en paramètre la grammaire (une liste de "Rules")

Un test global - *CheckAll* - se charge de tester l'ensemble de ces tests.

4.3 Test de symbole vide (*CheckLambda*)

Ce test s'assure qu'aucune règle n'a de symbole vide (lambda). Ce test parcourt simplement l'arbre de la définition grammaire et vérifie l'absence de terminaux lambda.

4.4 Test de conflit de préférence (*CheckPrecedence)*

Ce test vérifie qu'il n'y ait pas de conflit entre les précédence de symbole. Le seul conflit autorisé est entre '¡.' et '=.' qui donne la relation de précédence '¡.='.

Pour faire cela, le test construit deux ensemble pour chaque non-terminal : First et Last. First(A) contient l'ensemble des terminaux et non terminaux pouvant se trouver sur l'extrème gauche de A. Last(A) contient l'ensemble des terminaux et non terminaux pouvant se trouver sur l'extrème droide de A.

Pour calculer Tous les First on utilise l'algorithme suivant :

1. Pour tout non terminal A de la grammaire : On identifie les règles correspondantes. Pour chacune d'entre elles on identifie si elles commencent par un terminal, et si c'est le cas on les ajoute à First(A).
2. Pour tout non terminal A de la grammaire : On identifie les règles correspondantes, Pour chacune d'entre elles, on identifie si elles commencent par un non terminal B. Si c'est le cas on ajoute B et First(B) à First(A).
3. on répète l'étape précédente tant que les First changent.

L'algorithme pour trouver les Last est identique à ceci près qu'on examine les cotés droit des règles.

Une fois que l'on a ces deux ensembles et que l'on sait qu'il n'y a pas de symboles vide dans la grammaire on peut calculer la table de précedence, grace à l'algorithme suivant :

1. On initialise tous les éléments de la table à 'Nothing'
2. On identifie toutes les parties droites, et pour chacune d'entre elles on identifie tous les symboles cotes à cotes XY.
3. On ajoute X. =Y dans la table.
4. Si X non terminal : Pour tout symbole s dans Last(X), Pour tout terminal t dans First(Y) on met s. >t dans la table. Si Y est non-terminal on met s. >Y dans la table.
5. Si Y non terminal : Pour tout symbole s dans First(Y), on met X < . s dans la table.
6. Si à un moment on met une relation dans la table là ou précédemment il n'y avait pas 'Nothing', on a un conflit. Les conflits impliquant < . , =. et < . = sont résolus en insérant < . = dans la table. Les autres conflits sont marqués en tant que conflits et reportés.
7. Si il y a des conflits non résolus dans la table alors le Test échoue, la grammaire n'est pas valide Weak Precedence.

4.5 Test de Suffixe

Ce test vérifie que les relations de suffixe sont respectées. C'est à dire : Pour tout couple de règles $Z -> B$, $X -> AYB$, où A représente une suite de symbole, Y un symbole unique, et B et une suite non vide symboles, on vérifie que $!(Y .= X)$,$!(Y < . X)$ et $!(Y < . = X)$.

Pour ce faire nous utilisons la méthode suivante : Pour chaque règle $Z -> B$, pour toute autre règle $X -> Y$ on regarde si on peut matcher B comme suffixe de Y. Dans l'affirmative, on identifie le symbole juste avant le suffixe. Nous pouvons a ce moment vérifier la condition décrite expliquée précédemment. Si elle n'est pas vérifiée. le Test échoue, la grammaire n'est pas valide.

```
/**
 *
 * @param rules The rules list
 * @param table The precedence table
 * @return a list of tuple (triple) that contains the two conflicting rules and the
 */
public static List<RulesTuple> checkSuffix(List<Rule> rules, Hashtable<Term,Hashtabl
```

5 Test du vérificateur

Nous avons testé notre vérificateur de grammaire avec les grammaires fournie sur icampus pour finalement tester avec notre grammaire. Ici je ne montrerai que le test final pour notre grammaire.

Teste avec la première grammaire

```
A --> A a
A --> B
B --> a
B --> A B
```

réécrite en bnf pour notre parser :

```
<A> ::= <A> 'a'
<A> ::= <B>
<B> ::= 'a'
<B> ::= <A> <B>
```

```
Check conflit suffix :
```

```
----------------------------
A   A
<.
----------------------------
----------------------------
B   A
<=
----------------------------
conflit avec a:true
A ::= A:false a:true
B ::= a:true

conflit avec B:false
B ::= A:false B:false
A ::= B:false
```

Validity grammar : false
Ce test concorde bien avec l'exmple donné

```
Grammaire non WP :
règle 1 : B --> a
règle 2 : A --> A a
A <. B
Grammaire non WP :
règle 1 : A --> B
règle 2 : B --> A B
A <. A
```

Test avec une autre grammaire WP :

```
E --> T
E --> + T
E --> E + T
T --> F
T --> T * F
F --> x
F --> ( E )
```

qu'on a réécrit en :

```
<E> ::= <T>
<E> ::= '+' <T>
<E> ::= <E> '+' <T>
<T> ::= <F>
```

```
<T> ::= <T> '*' <F>
<F> ::= 'x'
<F> ::= '(' <E> ')'
```

C'est bien une grammaire WP :

```
Precedence Table
    |(   |)   |*   |+   |T   |E   |F   |x   |
     =======================================
(   | <. |    |    | <. | <. | <= | <. | <. |
     ---------------------------------------
)   |    | .> | .> | .> |    |    |    |    |
     ---------------------------------------
*   | <. |    |    |    |    |    | .= | <. |
     ---------------------------------------
+   | <. |    |    |    | <= |    | <. | <. |
     ---------------------------------------
T   |    | .> | .= | .> |    |    |    |    |
     ---------------------------------------
E   |    | .= |    | .= |    |    |    |    |
     ---------------------------------------
F   |    | .> | .> | .> |    |    |    |    |
     ---------------------------------------
x   |    | .> | .> | .> |    |    |    |    |
     ---------------------------------------

No conflicts in table
Validity precedence : true
Check conflit suffix :
Validity grammar : true
```

Dernière grammaire qui est normalement fausse :

```
E --> T
E --> + T
E --> T + E
T --> F
T --> F * T
F --> x
F --> ( E )

Precedence Table
    |(   |)   |*   |+   |T   |E   |F   |x   |
     =======================================
(   | <. |    |    | <. | <. | .= | <. | <. |
```

```
     ---------------------------------------
)    |    | .> | .> | .> |    |    |    |    |
     ---------------------------------------
*    | <. |    |    |    | .= |    | <. | <. |
     ---------------------------------------
+    | <. |    |    | <. | <= | .= | <. | <. |
     ---------------------------------------
T    |    | .> |    | .= |    |    |    |    |
     ---------------------------------------
E    |    | .= |    |    |    |    |    |    |
     ---------------------------------------
F    |    | .> | .= | .> |    |    |    |    |
     ---------------------------------------
x    |    | .> | .> | .> |    |    |    |    |
     ---------------------------------------
```

```
No conflicts in table
Validity precedence : true
Check conflit suffix :
----------------------------
E   +
.=
----------------------------
conflit avec T:false
E ::= +:true T:false
E ::= T:false

Validity grammar : false
```

Elle est bien fausse et la faute est la même que le test.

Maintenant que nous pouvons accorder un peu de crédit à notre vérificateur. Testons notre grammaire, en épargnant la table de précédence.

```
No conflicts in table
Validity precedence : true
Check conflit suffix :
Validity grammar : true
```