PCL1 : Rapport d'activité

Pierre GUYOT, Pierre-Yves JACQUIER, Arnaud KRAFFT, Titouan LANGLAIS

15 Janvier 2024

1 Introduction

Pour ce projet, nous avons décidé de programmer majoritairement en Java.

2 Grammaire

2.1 Transformation de la grammaire

Pour que la grammaire soit exploitable, il a fallu la modifier pour qu'elle soit LL(1). La première étape, la plus simple, a été de changer celle qui était donné pour avoir une grammaire plus exploitable. Il a donc fallu modifier toutes les règles où apparaissaient des terminaux ou des non terminaux de la forme $A^+_,$, $B?$ ou C^*. Nous avons donc ajouté des règles intermédiaires pour régler ces problèmes.

Par exemple, $A \to B?a$ devient:
$A \to Binterro$
$Binterro \to Ba$
$Binterro \to a$

De même, $A \to B^*C$ devient:
$A \to BetoileC$
$Betoile \to BBetoile$
$Betoile \to \epsilon$

Pour les règles de la forme $A^+_,$, nous avons choisi de transformer cela de la façon suivante:

$A \to B^+_,C$ devient:
$A \to BBetoileC$
$Betoile \to ,BBetoile$
$Betoile \to \epsilon$

2.2 Obtenir une grammaire LL(1)

Après avoir obtenu une grammaire sur laquelle on pouvait déterminer si elle était (ou non) LL(1), nous avons pu attaquer la partie épineuse du problème. Pour cela nous nous somme aidés de l'outil Grammophone, permettant de signaler les problèmes faisant que la grammaire n'est pas LL(1). En particulier, cet outil nous a indiqué si tous les non-terminaux étaient atteignables et réalisables, si la grammaires contenait des cycles ou si la grammaire était ambiguë.

Cet outil nous a directement permis de savoir si notre grammaire était LL(1) et, le cas échéant, nous a montré les non-terminaux posant problème. Cet outil nous a donc été d'une grande aide pour rendre la grammaire LL(1) petit à petit.

Le plus gros problème que nous avons eu a été avec les règles:

$decl \rightarrow type\, IDF$;
$decl \rightarrow type\, IDF\, is\, access\, IDF$;
$decl \rightarrow type\, IDF\, is\, record <champs>^{+}\, end\, record$;
$type \rightarrow IDF$
(⟨ident⟩a été remplacé par IDF)

Ces règles créent un conflit. Nous avons réglé ce conflit en rassemblant les deux règles issues de decl en une seule grâce à un nouveau non terminal que nous avons appelé ISinterro. Ainsi, ces règles deviennent:

$decl \rightarrow type\, IDF\, ISinterro$;
$ISinterro \rightarrow is\, ACCInterro$
$ISinterro \rightarrow \epsilon$
$ACCInterro \rightarrow access\, IDF$
$ACCInterro \rightarrow record\, IDF\, IDFEtoile\, :\, ACCESSInterro\, IDF\, ;\, TYPEInterro\, end\, record$
$type \rightarrow IDF$

Les autres règles qui ont posé problème sont les règles partant de ⟨expr⟩. Tout d'abord, à cause de la récursivité gauche causée par la règle:

$<expr> \rightarrow <expr> <opérateur> <expr>$

Ainsi que par les règles:

$<expr> \rightarrow <accès>$
$<accès> \rightarrow <expr>.<ident>$
Ces conflits ont été les plus complexes à résoudre et ils nous ont forcés à retravailler entièrement la partie de la grammaire liée aux expressions. Nous avons donc dû séparer ⟨expr⟩en un grand nombre de règles, avec un ou deux non terminaux par opérateur. Par exemple, pour OR, on a:

$OR \rightarrow AND\, ORPrime$
$ORPrime \rightarrow or\, OR$
$ORPrime \rightarrow orelse\, OR$
$ORPrime \rightarrow \epsilon$

On a donc des règles similaires pour tous les opérateurs. L'ordre dans lequel ces règles ce trouvent est prédéterminé par le tableau des associativités et des précédences donné dans le sujet.

2.3 Grammaire finale

La grammaire que nous avons obtenue à la fin est bel et bien LL(1), mais pas sans faille. Elle a toujours un problème que nous n'avons pas réussi à retirer: elle penche à droite. Ceci est dû à la façon dont nous avons écrit les opérateurs.

3 Analyse Lexicale - Lexeur

3.1 Lecteur

Le lexeur est composé de plusieurs parties, sa première étant le lecteur. Ce dernier est un simple lecteur de fichier textuel qui lit caractère par caractère le fichier de code source. Il ne conserve que les deux derniers caractères qui viennent d'être lus. De plus, il gère les premières erreurs de caractères dans le cadre de caractères non ASCII.

3.2 Structure de données

Le lexeur utilise une classe nommée Mot_L permettant de grouper les caractères en un string, tout en gardant l'emplacement du mot dans le code source (numéro de ligne et de caractères). De plus, une fois que le mot est reconnu, un code lui est attribué dans cette structure. En plus de cela, nous utilisons une HashMap pour stocker tous les codes associés aux unités lexicales. Des unités lexicales peuvent être ajoutées sans que cela pose de souci pour le reste du lexeur. Il faut simplement indiquer en plus via des variables statiques le code pour les identifiants.

3.3 Test des mots

Le lexeur contient aussi une classe is_Char permettant de tester les différents types de caractères (symboles, alphanumériques etc) ainsi que de donner le code des unités correspondantes à certaines chaînes de caractères. Cette classe applique aussi une expression régulière afin de savoir si on a affaire à un identifiant (variable, constante entière ou un caractère).

3.4 Analyseur

Comme la grammaire possède des unités syntaxiques spéciales comme "or else" ou bien "and then" ou encore les constantes caractères avec comme exemple 'a', il faut pouvoir les distinguer d'une simple unité lexicale. L'analyseur permet

alors de garder en mémoire jusqu'à 3 unités lexicales, permettant d'être sûr de ne pas se tromper en unité lexicale simple ou composée de plusieurs sous unités.

3.5 Écrivain

Une fois que l'on a reconnu une unité lexicale via l'analyseur, on l'écrit dans un nouveau fichier via l'écrivain. Ce dernier écrit le code dans le fichier en un caractère UTF-8 lié au code du caractère. Si c'est un identifiant, alors le code de l'identifiant est écrit juste après dans le fichier. Ce fichier a comme extension .lex. Le nom des variables est sauvegardé dans un autre fichier sauvegardé en .idf. Il est généré à la fin de la lecture et de l'analyse du fichier de code source, contrairement au fichier .lex qui est généré au fil de la lecture.

3.6 Gestion des erreurs

Si les différentes parties de ce programme rencontrent une erreur, elle est signalée dans la sortie standard et elle est remontée à l'analyseur lexical pour qu'il puisse signaler au parseur de ne pas analyser syntaxiquement le code. Pour autant, l'analyse lexicale continue pour détecter toutes les potentielles erreurs et les soumettre à l'utilisateur. Un numéro de ligne et de caractère est gardé pour pouvoir signaler à l'utilisateur où se trouve précisément l'erreur. De plus, l'écrivain garde la structure des lignes pour pouvoir savoir sur quelle ligne se produit une erreur dans l'analyseur syntaxique.

4 Analyse Syntaxique - Sémantique

4.1 Importation de la grammaire

Pour importer la grammaire dans notre programme, nous avons décidé d'utiliser l'outil Grammophone pour nous donner la table d'analyse. À partir de notre grammaire que nous avons rendu LL(1) (expliqué dans la première partie), l'outil nous sort une table d'analyse en HTML. Nous avons alors créé un parseur HTML qui permet de l'obtenir en CSV. Ce CSV sera la base pour la suite de notre analyse syntaxique.

4.2 Lecture et exploitation de la table d'analyse (CSV)

La lecture de la table d'analyse en fichier CSV permet de donner à notre analyseur syntaxique la connaissance de tous les terminaux et non-terminaux qu'il va utiliser. L'analyseur syntaxique les stocke donc tous dans deux hashmaps différentes. Notre parseur de CSV crée ensuite une table d'analyse comportant toutes les règles liées à des codes, permettant d'avoir une structure de données exploitable pour l'analyse syntaxique.

4.3 Analyse

L'analyse commence d'abord par une lecture du fichier .lex, permettant d'avoir une tête de lecture que l'on déplacera au cours de l'analyse. On rajoute à cela la lecture du fichier .idf pour pouvoir inclure les références dans l'AST. L'analyseur est codé sous la forme d'un petit algorithme utilisant une pile, une tête de lecture et la table d'analyse. À chaque itération, on enlève un élément de la pile. Si l'élément est un terminal, on vérifie que c'est le même terminal que la tête de lecture, puis on avance la tête de lecture. Si l'élément est un non-terminal, on lit dans la table d'analyse la règle se situant à la ligne du numéro du non-terminal de la pile, et à la colonne du terminal dans la tête de lecture. Si la règle est correcte, on rajoute les terminaux et non terminaux de la règle à la pile (sans oublier d'inverser l'ordre). L'analyse syntaxique est correcte si la pile est vide, (et la tête de lecture sera donc elle aussi vide). L'analyse syntaxique est incorrecte si le terminal de la pile est différent de la tête de lecture, ou si la règle sélectionné dans la table d'analyse est inexistante.

4.4 Gestion des erreurs

Nous n'avons pas réussi à continuer l'analyse syntaxique si une erreur survenait. En effet, nous ne pouvions pas connaître les raisons de l'erreur du code (omission ou ajout d'une ou plusieurs unités lexicales). Ainsi, faire la supposition d'une certaine erreur pourrait conduire à une cascade d'erreurs rendant la gestion d'erreur illisible. Nous avons donc préféré nous arrêter à la première erreur syntaxique survenue dans le programme, tout en signalant ce qui la cause, et la ligne à laquelle elle apparaît.

5 Arbre Syntaxique

5.1 Création de l'arbre de dérivation

L'arbre de dérivation se récupère automatiquement de la même manière que l'on crée la pile. Pour créer l'arbre de dérivation, on utilise une classe noeud que l'on découpe en deux sous-classes : noeud terminal et noeud non terminal. On utilise deux piles entre les noeuds terminaux et non terminaux pour pouvoir remplir l'arbre de dérivation au cours de l'analyse syntaxique.

5.2 Élagage de l'arbre

L'élagueur de notre arbre de dérivation fait plusieurs choses. Tout d'abord, il supprime les noeuds terminaux représentant des symboles dont la présence dans l'arbre n'est pas nécessaire, par exemple '(', ';' ne servent pas dans l'arbre, car les noeud non terminaux qui les contiennent impliquent leur existence. En suite, il supprime tous les noeuds n'ayant aucun enfant. Après, il supprime tous les noeuds n'ayant qu'un enfant en transférant leur enfant à leur parent. En suite, il "compresse" les noeuds en cascade (que l'on appelle étoiles et primes).

5.3 Représentation Graphique

Afin de représenter nos arbres de manière plus graphique, nous avons décidé d'utiliser le moteur graphique de plantUML. Ainsi, nous avons une fonction permettant de transformer nos structures de données Java en un fichier plantUML, directement utilisable sur le site web de plantUML ou avec des extensions sur VS Code directement.

5.4 Forme finale

L'arbre syntaxique en sortie de l'élagueur n'est pas encore sous sa forme finale. La conversion d'une forme à l'autre sera réalisée lors de la partie 2 du module, afin d'implémenter des contrôles syntaxiques.

6 Gestion de projet

6.1 répartition des tâches

La répartition des tâches a été la suivante: Arnaud et Titouan ont été responsables de la grammaire. Ils l'ont transformée pour la rendre LL(1) et l'ont modifiée pour corriger divers problèmes tout au long du projet. Pierre-Yves s'est concentré dès le début du projet sur le code java permettant de générer des arbres syntaxiques abstraits. Pierre s'est occupé du Parseur et du Lexeur. Titouan a également rédigé des programmes en Ada afin de faire des tests dessus.

7 Exemples

7.1 HelloWorld

Le code :

```
1  with Ada.Text_IO;use Ada.Text_IO;
2
3  procedure Hello_World is
4  begin
5     put('H')
6     -- put('e')
7  end Hello_World;
```

L'analyseur Lexical :

```
[DEBUG] Mot écrit : with - code : 1
[DEBUG] Ligne : 1 - Caractère : 1
[DEBUG] Mot écrit : Ada - code : 2
[DEBUG] Ligne : 1 - Caractère : 6
[DEBUG] Mot écrit : . - code : 3
[DEBUG] Ligne : 1 - Caractère : 9
[DEBUG] Mot écrit : Text_IO - code : 2
[DEBUG] Ligne : 1 - Caractère : 10
[DEBUG] Mot écrit : ; - code : 4
[DEBUG] Ligne : 1 - Caractère : 17
[DEBUG] Mot écrit : use - code : 5
[DEBUG] Ligne : 1 - Caractère : 18
[DEBUG] Mot écrit : Ada - code : 2
[DEBUG] Ligne : 1 - Caractère : 22
[DEBUG] Mot écrit : . - code : 3
[DEBUG] Ligne : 1 - Caractère : 25
[DEBUG] Mot écrit : Text_IO - code : 2
[DEBUG] Ligne : 1 - Caractère : 26
[DEBUG] Mot écrit : ; - code : 4
[DEBUG] Ligne : 1 - Caractère : 33
[DEBUG] Mot écrit : procedure - code : 6
[DEBUG] Ligne : 3 - Caractère : 1
[DEBUG] Mot écrit : Hello_World - code : 2
[DEBUG] Ligne : 3 - Caractère : 11
[DEBUG] Mot écrit : is - code : 7
[DEBUG] Ligne : 3 - Caractère : 23
[DEBUG] Mot écrit : begin - code : 8
[DEBUG] Ligne : 4 - Caractère : 1
[DEBUG] Mot écrit : put - code : 2
[DEBUG] Ligne : 5 - Caractère : 4
[DEBUG] Mot écrit : ( - code : 19
[DEBUG] Ligne : 5 - Caractère : 7
[DEBUG] Mot écrit : 'H' - code : 48
[DEBUG] Ligne : 5 - Caractère : 10
[DEBUG] Mot écrit : ) - code : 20
[DEBUG] Ligne : 5 - Caractère : 11
[DEBUG] Mot écrit : end - code : 9
[DEBUG] Ligne : 7 - Caractère : 1
[DEBUG] Mot écrit : Hello_World - code : 2
[DEBUG] Ligne : 7 - Caractère : 5
[DEBUG] Mot écrit : ; - code : 4
[DEBUG] Ligne : 7 - Caractère : 16
[DEBUG] Mot écrit : eof - code : 10
[DEBUG] Ligne : 9 - Caractère : 1
```

Le fichier .lex :

ogrammes > Fonctionne > hello_world.adb.lex

```
1   ÉÊÉËÊÊÌÍÊÉËÊÊÌ&&ÎÊËÏ&Ð&ÊÌÛøÐÜ&&ÑÊËÌÒ
```

Le fichier des idf :

```
1=ada;
2=text_io;
3=hello_world;
4=put;
```

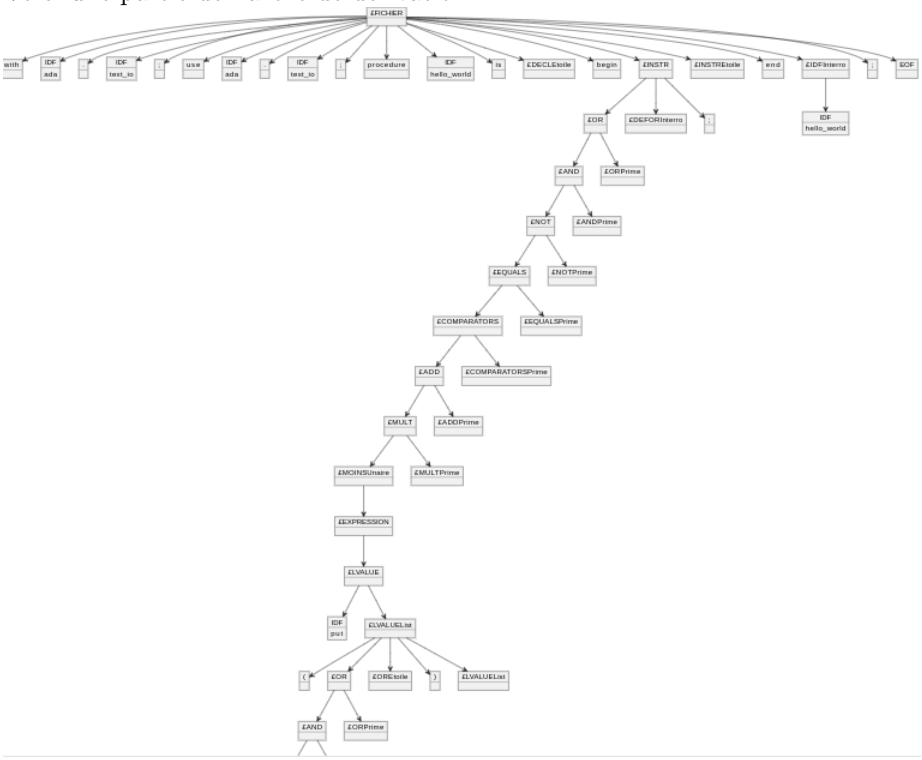
Début de l'analyseur syntaxique :

```
[DEBUG] Element Pile Non-Terminal : £FICHIER code : -1
[DEBUG] Tete : with code : 1
[DEBUG] Element Pile Terminal : with code : 1
[DEBUG] Tete : with code : 1
[DEBUG] Element Pile Terminal : IDF code : 2
[DEBUG] Tete : IDF code : 2
[DEBUG] Element Pile Terminal : . code : 3
[DEBUG] Tete : . code : 3
[DEBUG] Element Pile Terminal : IDF code : 2
[DEBUG] Tete : IDF code : 2
[DEBUG] Element Pile Terminal : ; code : 4
[DEBUG] Tete : ; code : 4
[DEBUG] Element Pile Terminal : use code : 5
[DEBUG] Tete : use code : 5
[DEBUG] Element Pile Terminal : IDF code : 2
[DEBUG] Tete : IDF code : 2
[DEBUG] Element Pile Terminal : . code : 3
[DEBUG] Tete : . code : 3
[DEBUG] Element Pile Terminal : IDF code : 2
[DEBUG] Tete : IDF code : 2
[DEBUG] Element Pile Terminal : ; code : 4
[DEBUG] Tete : ; code : 4
[DEBUG] Element Pile Terminal : procedure code : 6
[DEBUG] Tete : procedure code : 6
[DEBUG] Element Pile Terminal : IDF code : 2
[DEBUG] Tete : IDF code : 2
```

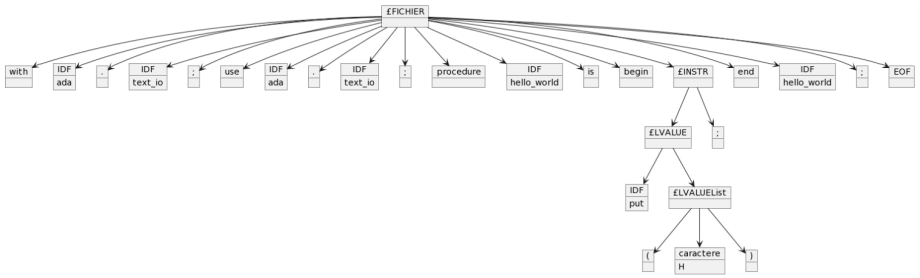
Erreur car le code n'a pas de point virgule à la fin de la ligne avec l'instruction put :

```
[DEBUG] Tete : ) code : 20
[DEBUG] Element Pile Non-Terminal : £NOTPrime code : -27
[DEBUG] Tete : ) code : 20
[DEBUG] Element Pile Non-Terminal : £ANDPrime code : -25
[DEBUG] Tete : ) code : 20
[DEBUG] Element Pile Non-Terminal : £ORPrime code : -23
[DEBUG] Tete : ) code : 20
[DEBUG] Element Pile Non-Terminal : £OREtoile code : -14
[DEBUG] Tete : ) code : 20
[DEBUG] Element Pile Terminal : ) code : 20
[DEBUG] Tete : ) code : 20
[DEBUG] Element Pile Non-Terminal : £LVALUEList code : -39
[DEBUG] Tete : end code : 9
[ERROR] Erreur : règle vide : 39 - 9 - ligne : 7
```

Voici une partie de l'arbre de dérivation :



Et voici l'arbre de dérivation élagué :

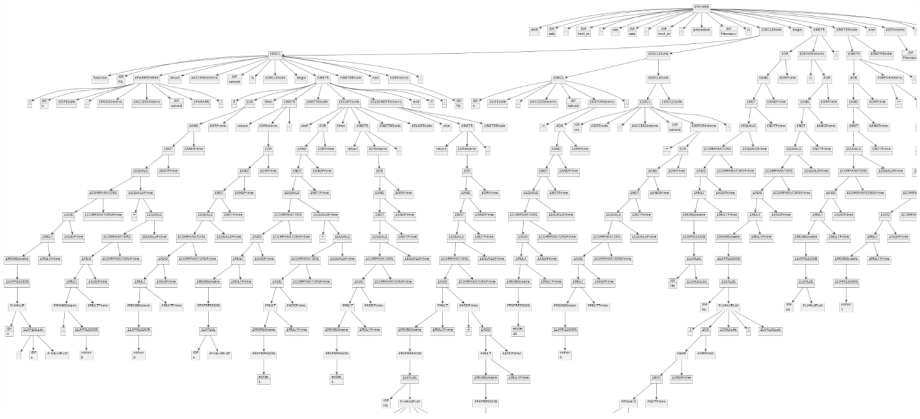


7.2 Fibonaccii

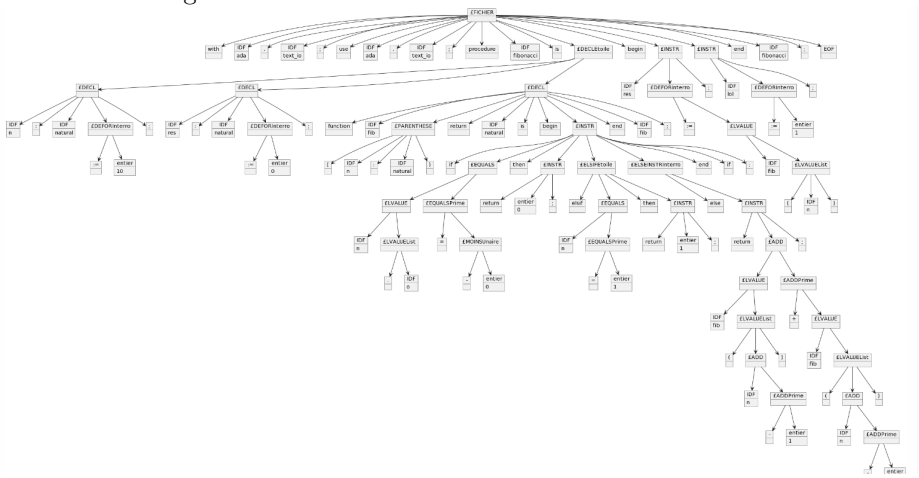
Voici le code :

```
1   with Ada.Text_IO; use Ada.Text_IO;
2
3   procedure Fibonacci is
4       function Fib(n: Natural) return Natural is
5       begin
6           if n.o = -0 then
7               return 0;
8           elsif n = 1 then
9               return 1;
10          else
11              return Fib(n - 1) + Fib(n - 2);
12          end if;
13      end Fib;
14
15      --Variables
16  n : Natural := 10;
17  res : Natural := 0;
18
19      --Main
20  begin
21      res := Fib(n);
22      lol := 1;
23      --put("Fibonacci de " & n'Image);
24      --put(" est :" & res'Image); 'aB'
25  end Fibonacci;
```

Voici l'arbre de dérivation



Voici l'arbre élagué :



7.3 Erreurs dans le lexeur

Voici un exemple lorsque plusieurs erreurs sont dans le lexeur. Le code en question :

```
1  put('é');
2  _Var_impossible
3  1var_impossible
4  var1_possible
5  -- put('e')
6  != test
```

Et la sortie du lexeur

```
[ERROR] Le fichier contient des caractères non-ASCII à la ligne 1 et au caractère 6
[DEBUG] Mot écrit : put - code : 2
[DEBUG] Ligne : 1 - Caractère : 1
[DEBUG] Mot écrit : ( - code : 19
[DEBUG] Ligne : 1 - Caractère : 4
[DEBUG] Mot écrit : ' - code : 50
[DEBUG] Ligne : 1 - Caractère : 5
[DEBUG] Mot écrit : ' - code : 50
[DEBUG] Ligne : 1 - Caractère : 7
[ERROR] Caractère non reconnu à la ligne 2 et au caractère 1 : _
[ERROR] Erreur : mot non reconnu à la ligne 2 et au caractère 1 :
[DEBUG] Mot écrit : ) - code : 20
[DEBUG] Ligne : 1 - Caractère : 8
[DEBUG] Mot écrit : ) - code : 20
[DEBUG] Ligne : 1 - Caractère : 8
[DEBUG] Mot écrit : ; - code : 4
[DEBUG] Ligne : 1 - Caractère : 9
[DEBUG] Mot écrit : Var_impossible - code : 2
[DEBUG] Ligne : 2 - Caractère : 2
[DEBUG] Mot écrit : 1 - code : 52
[DEBUG] Ligne : 3 - Caractère : 1
[ERROR] Caractère non reconnu à la ligne 6 et au caractère 1 : !
[ERROR] Erreur : mot non reconnu à la ligne 6 et au caractère 1 :
[DEBUG] Mot écrit : var_impossible - code : 2
[DEBUG] Ligne : 3 - Caractère : 2
[DEBUG] Mot écrit : var_impossible - code : 2
[DEBUG] Ligne : 3 - Caractère : 2
[DEBUG] Mot écrit : var1_possible - code : 2
[DEBUG] Ligne : 4 - Caractère : 1
[DEBUG] Mot écrit : = - code : 37
[DEBUG] Ligne : 6 - Caractère : 2
[DEBUG] Mot écrit : test - code : 2
[DEBUG] Ligne : 6 - Caractère : 4
[DEBUG] Mot écrit : eof - code : 10
[DEBUG] Ligne : 8 - Caractère : 1
```