Implémentation d'un compilateur simplifié

Federico Pfeiffer

11 juin 2017

Résumé

Ce rapport décrit l'implémentation d'un compilateur du langage Hepial, défini les spécifications en annexe.

Table des matières

1 Présentation

1.1 État général de l'implémentation

Le compilateur a pu être implémenté selon le cahier des charges demandé : il analyse syntaxiquement un code source en créant un arbre abstrait et une table des symbole, puis recherche les erreurs sémantiques possibles au sein de l'arbre abstrait et la table des symboles. Si aucune erreur a été détectée, le compilateur produit un code source en jasmin, qui est ensuite compilé en byte code .class.

La notion de récursivité de fonctions, et de portée des variables ont également pu être implémentée. De même, des expressions complexes telles que celles-ci ont également pu être implémentées. *if(a < b || 2*6+4 > 0)*. Enfin, des imbrications de boucles (telles que while et for) sont également possible.

Bien que les tableaux aient été ajoutés, ceux-ci n'ont été que très peu testés. Plusieurs dimensions sont en théories possibles, mais cela n'a pas été testé non plus. La gestion des tableaux a été définie de la sorte : array[5 .. 10] crée un tableau de 5 cases, dont l'index va de 0 à 5.

Enfin, l'expression *inégal* s'écrit *!=* au lieu de *<>*. Par ailleurs, la déclaration de fonction au sein d'une fonction est interdite par soucis de simplicité.

1.2 Utilisation

L'utilisation du compilateur se fait de la manière suivante :

1. Compilez le compilateur avec la commande *make*. Cela va générer un dossier */bin/* contenant le byte code du compilateur. L'exécutable nommé *hepiaCompile* sera généré dans le même dossier où se trouve le fichier *make*. (Pas besoin de changer de dossier avec *cd*).
2. Compilez ensuite le fichier hepial avec la commande *bash hepiaCompile* suivit du nom du fichier Hepial à compiler. Cela va générer un dossier *compiledBin* contenant le byteCode du programme Hepial, ainsi que ses sources en jasmin. L'exécutable du programme sera généré dans le même dossier où se trouve le fichier *make*.

Extrait de code 1 – Utilisation du compilateur

```
# compiler le compilateur
$ make

# compiler le fichier hepial
$ bash hepiaCompile <fileName>  # fileName = input.txt par défaut

# lancer l exécutable compilé
$ bash <programName>
```

1.3 Vue d'ensemble

La compilation s'exécute dans cet ordre :

1. Génération de l'arbre abstrait, de la table des symboles pendant le parsing syntaxique de Cup.
2. Vérification sémantique dans l'arbre abstrait et dans la table des symboles.
3. Production du code : fichiers jasmin et fichier .class.

Les diverses étapes de la compilation sont exécutées depuis le fichier *HepialCompilateur.java*, à l'intérieur de la fonction *main*.

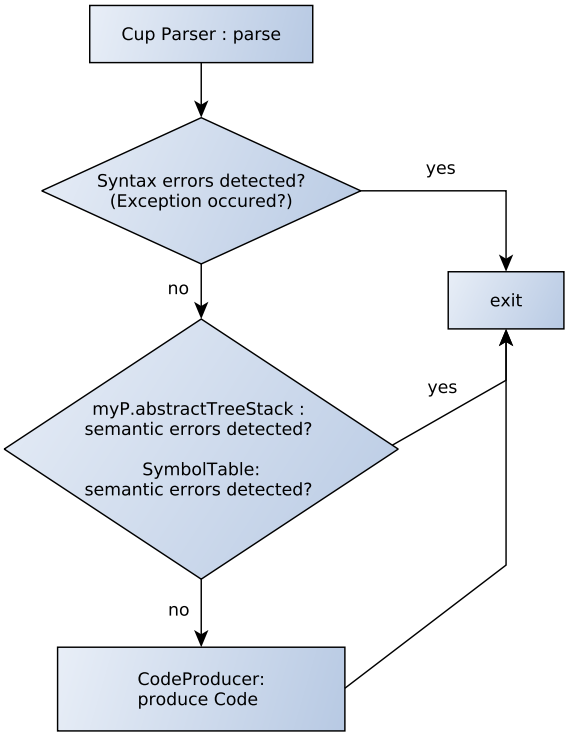

FIGURE 1 – Déroulement général du programme (fonction main())

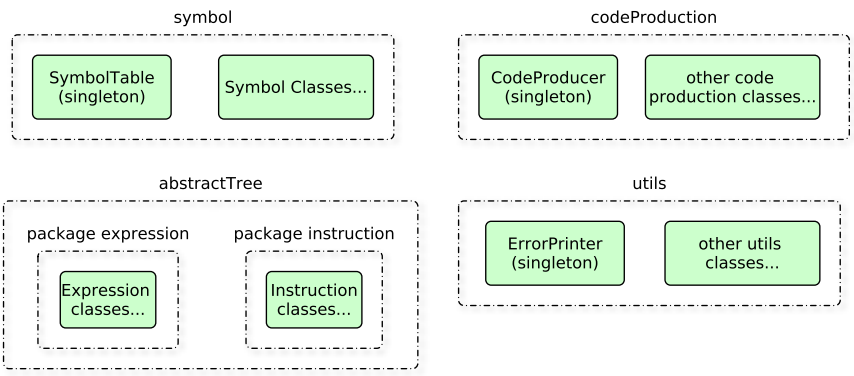

FIGURE 2 – Vue générale des packages et des principales classes

2 Construction de l'arbre abstrait et table des symboles

2.1 Arbre abstrait

L'arbre abstrait et la Table des Symboles sont générés pendant le parsing CUP, dans la première partie de la compilation.

Les expression et instructions héritent de la classe AsbtractTree. Les expressions et instructions détectées pendant le parsing CUP sont transmises de règle en règle à travers la pile de l'arbre abstrait, en empilant/dépilant les éléments de l'arbre abstrait. L'arbre abstrait est défini dans une des entêtes CUP.

Extrait de code 2 – Déclaration de la pile de l'arbre abstrait

```
init with {:
  abstractTreeStack = new Stack<AbstractTree>();
:};
```

À la fin du parsing, on obtient un seul élément dans la pile d'arbre abstrait : un BlocInstruction contenant toutes les instructions du programme. Les instructions des fonctions se trouvent dans le symbol FunctionSymbol, ajouté dans la table des symboles.

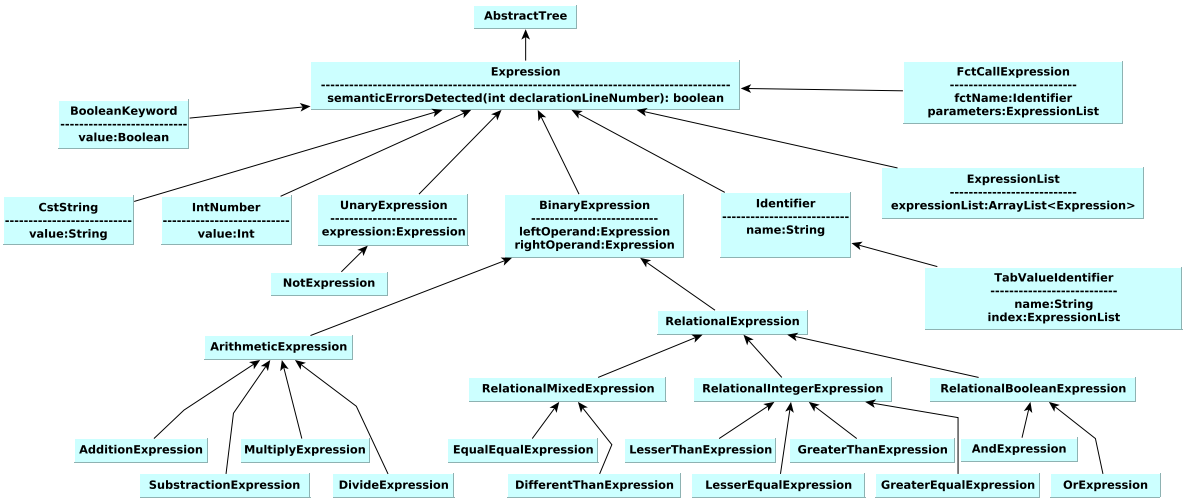

FIGURE 3 – Architecture des classes d'expressions

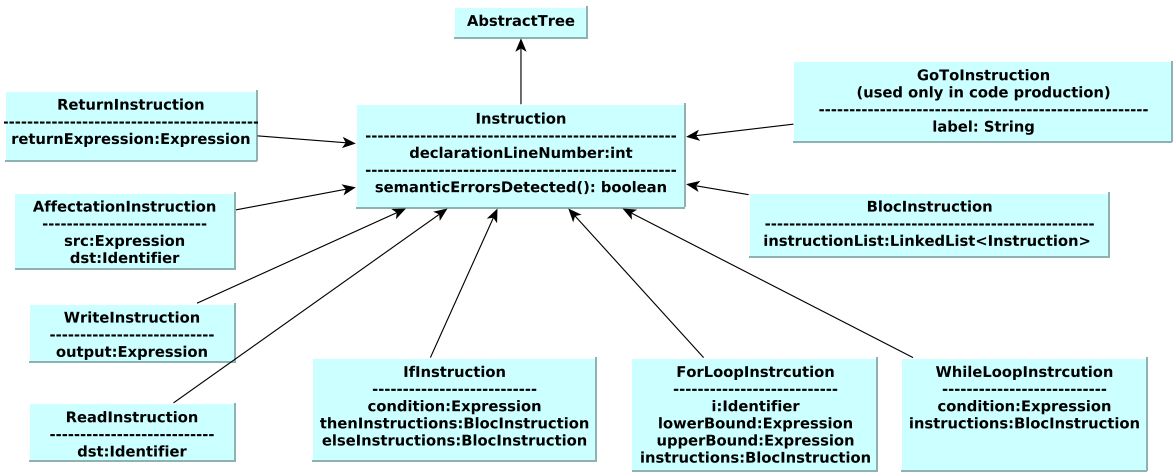

FIGURE 4 – Architecture des classes d'instructions

2.2 Table des symboles

Les déclarations (telles que déclaration de fonction ou de variables) détectées pendant le parsing cup sont transmises de règles en règles à l'aide du RETURN de chaque règle. Elles sont transmises en tant que *ArrayList<VarName, Symbol>*, ou plus précisément *ArrayList<SimpleEntry<String, Symbol> >*.
Ces éléments sont ensuite insérés dans la table des symboles par la règle adéquate.
La table des symboles, de classe *SymbolTable* est un singleton qui peut être accédée de manière statique à n'importe quel endroit du programme.

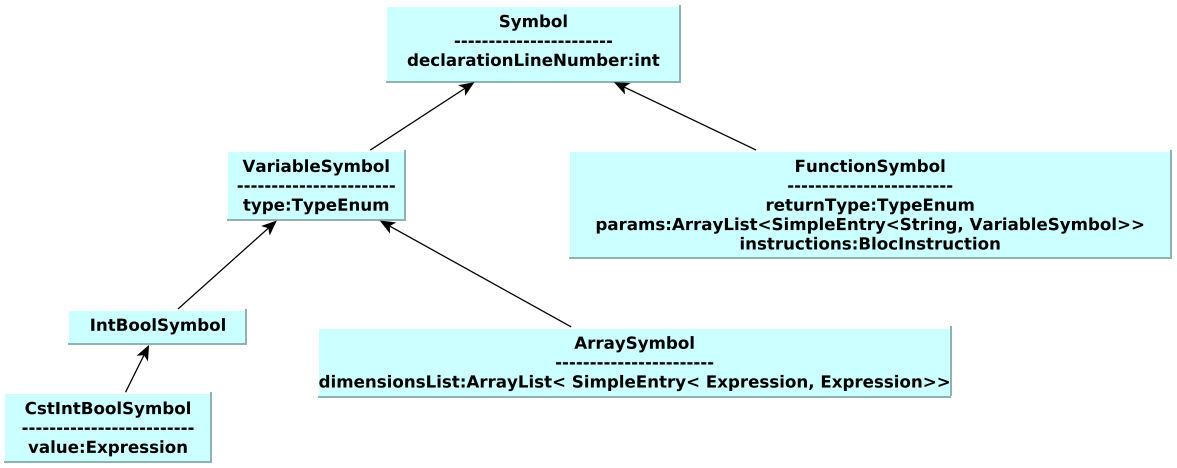

FIGURE 5 – Architecture des classes de symboles

2.3 Vérification syntaxique

La vérification syntaxique se fait par CUP directement. Si une erreur de syntaxe est détectée, une exception est levée et le compilateur affiche "parse error".

3 Vérification sémantique

3.1 Pattern utilisé

La vérification sémantique se fait une fois le parsing cup effectué. Nous nous retrouvons avec une tables des symboles complète, et une pile d'arbre abstrait contenant uniquement un bloc d'instructions *(BlockInstruction)*.

A partir de là, le plus simple pour moi était d'utiliser le pattern *Controller* défini dans les documents de cours. Si j'ai bien compris, il se rapproche du pattern *Template method pattern* défini sur Wikipedia. Chaque classe de l'arbre abstrait comporte la méthode *semanticErrorsDetected()*. Cette fonction vérifie si l'instance comporte des erreurs syntaxiques, et appelle la fonction *semanticErrorsDetected()* de tous les attributs de type arbre abstrait que l'instance contient. Elle renvoie faux si une erreur a été détectée soit au sein de l'instance, soit au sein d'un de ses attributs.

Par exemple, la fonction *WhileLoopInstruction.semanticErrorsDetected()* va vérifier que son attribut *WhileLoopInstruction.condition* est bien de type booléen. La fonction va ensuite appeler *WhileLoopInstruction.condition.semanticErrorsDetected()* et *WhileLoopInstruction.instructions.semanticErrorsDetected()*.

Le même pattern est utilisé pour la vérification sémantique au sein des symboles. La table des symboles contient elle aussi une fonction *semanticErrorsDetected()* afin de vérifier les erreurs sémantique au sein de tous les symboles déclarés.

3.2 Processus

Une fois ces méthodes définies, la vérification sémantique dans tout l'arbre abstrait, et dans tous les symboles peut être effectuée facilement depuis la fonction main, une fois le parsing syntaxique effectué.

Extrait de code 3 – Processus de vérification syntaxique

```
// check for semantic errors and print errors if any
boolean errorsDetected = false;
if(SymbolTable.getInstance().semanticErrorsDetected()){
  errorsDetected = true;
}
if(abstractTreeElement.semanticErrorsDetected()){
  errorsDetected = true;
}

if(errorsDetected){
     ErrorPrinter.getInstance().printErrors();
     System.exit(1);
}
```

3.3 Affichage des erreurs

L'instance *ErrorPrinter*, un singleton, permet aux fonctions *semanticErrorsDetected()* d'y logger les erreurs trouvées lors de la vérifiacation sémantique. Une fois la vérification sémantique terminée, on peut appeler la fonction *ErrorPrinter.getInstance().printErrors()* pour afficher les erreurs une à une. Cela permet d'effectuer une vérification sémantique complète, tout en loguant des erreurs, sans avoir à stopper la vérification sémantique à chaque fois qu'une erreur est trouvée.

4 Production du code

La production du code consiste en produire une représentation du programme en fichiers jasmin, pour être ensuite converti en byteCode java, executable par la JVM.

4.1 Architecture de base d'un programme Hepial

Le code ayant pour finalité d'être décrite comme du byteCode java, il parait approprié d'illustrer l'architecture d'un programme Hepial en une représentation java.

Une première classe est créée, contenant la méthode *static main()*. Cette classe est nommée avec le nom du programme définit dans l'entête du code Hepial . La méthode *static main()* sert uniquement à lancer une deuxième instance, *MainBlock*, contenant comme attributs les variables définies dans le bloc principal. Les instructions du bloc principal sont définis dans la fonction *MainBlock.mainFunction()*.

Extrait de code 4 – Architecture de base d'un programme Hepial

```
public class HepialProgramName{
  public static void main (String[] arg){
    MainBlock mainBlock = new MainBlock();
    mainBlock.mainFunction();
  }
}

public class MainBlock{
  // variables defined in main block go here

  public void mainFunction(){
      // main instructions go here
  }
}
```

Lors de la production du code, les fichier jasmin et .class du programme sont créées dans un dossier *compiledBin*. L'exécutable en tant que tel du programme Hepial est un fichier bash créé dans le dossier parent de *compiledBin*. Il se présente sous cette forme :

Extrait de code 5 – Exécutable programme Hepial

```
#/bin/sh
java -cp compiledBin HepialProgramName
```

Cette architecture a été choisie pour répondre aux enjeux définis dans la prochaine section.

4.2 Enjeux

Une première difficulté fut de trouver un moyen relativement simple de debugger les fichiers jasmin produits. L'utilisation des locales de la JVM ne permettaient pas de savoir quelle variable était réellement appelée. Appeler un attribut (une variable) par son nom se révéla beaucoup plus pratique pour debugger. Je me suis donc penché sur l'utilisation de classes java afin de représenter les variables déclarées, comme attributs d'une classe.

La notion de portée des variables entre les fonctions fut également un enjeu majeur. La possibilité de créer des variables locales à la fonction déclarée, tout en pouvant appeler des variables déclarées hors de cette fonction était un deuxième argument pour utiliser des classes java pour chaque Bloc : Une classe pour le bloc principal, et une classe par fonction est une solution viable que j'ai choisi pour répondre à la notion de portée des variables.
Chaque fonction créée peut être représentée sous forme d'une classe, dont le constructeur est appelé avec *MainBlock* comme référence.

Un dernier enjeu était la notion de récursivité : une fonction pouvant s'appeler elle même, il fallait trouver un moyen de garder la portée des variables intègre.

Par conséquent, j'ai choisi l'architecture suivante pour représenter les fonctions d'un programme Hepial.

4.3 Architecture d'une fonction

La représentation d'une fonction en jasmin/byteCode est la même que la représentation du *MainBlock*, à la différence que le constructeur de la fonction a comme paramètre la référence au *MainBlock* afin d'avoir accès aux variables de celui-ci.

Dans la classe fonction, des attributs supplémentaires sont ajoutés aux variables déclarées dans la fonction : les paramètres de la fonction elle même.

Afin d'illustrer ceci, prenons ce code Hepial de la figure suivante comme exemple. On crée une variable globale égale à 1. On définit une fonction prenant en paramètre un entier. Cette fonction a une variable locale. La fonction prends la valeur entré en paramètre, fait la somme entre celle-ci et la variable globale, puis affecte le résultat dans sa variable locale. Elle retourne ensuite la valeur de sa variable locale.

Extrait de code 6 – Architecture d'une fonction et de son appel

```
programme hepialProgramName

constante entier globale = 1;
entier resultat;
entier maFct(entier param1)
   entier locale;
   debutfonc
      locale = param1 + globale;
      retourne locale;
   finfonc
debutprg
   resultat = maFct(2);
finprg
```

La version java du programme Hepial aura donc cette forme :

Extrait de code 7 – Architecture de base d'un programme Hepial

```
public class MainBlock{
  public int globale;
  public int resultat;

  public void mainFunction(){
      MaFct maFct = new MaFct(this);
      resultat = maFct.mainFunction(2);
  }
}

public class MaFct{
  MainBlock mainBlock;
  public int locale;

  MaFct(MainBlock mainBlock){
     this.mainBlock = mainBlock;
  }

  int mainFunction(int param1){
      locale = this.mainBlock.globale + param1;
  }
}
```

4.4 Principales instances de la production du code

4.4.1 CodeProducer

La production du code est appelée depuis la fonction main du compilateur, à l'aide de l'appel suivant : *CodeProducer.getInstance().produceProgram()*. Cette fonction peut être représentée selon le schéma ci-dessous :

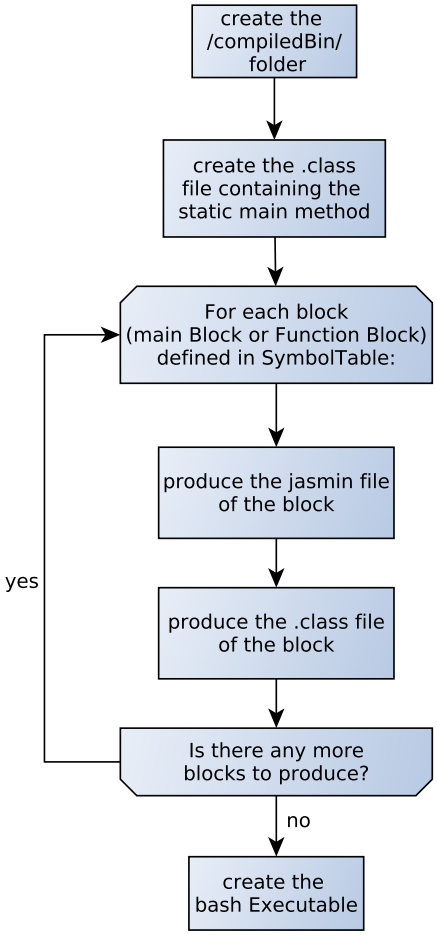

FIGURE 6 – Vue générale de la production de code

4.4.2 Block

Afin de créer les fichiers jasmin, *CodeProducer* s'aide de la classe *Block*. Cette dernière contient l'API nécessaire pour créer un fichier jasmin représentant chaque bloc du programme Hepial : à savoir le bloc principal, et les blocs représentant les fonctions déclarées.

Le schéma suivant illustre la manière dont la classe *CodeProducer* crée les fichiers jasmin :

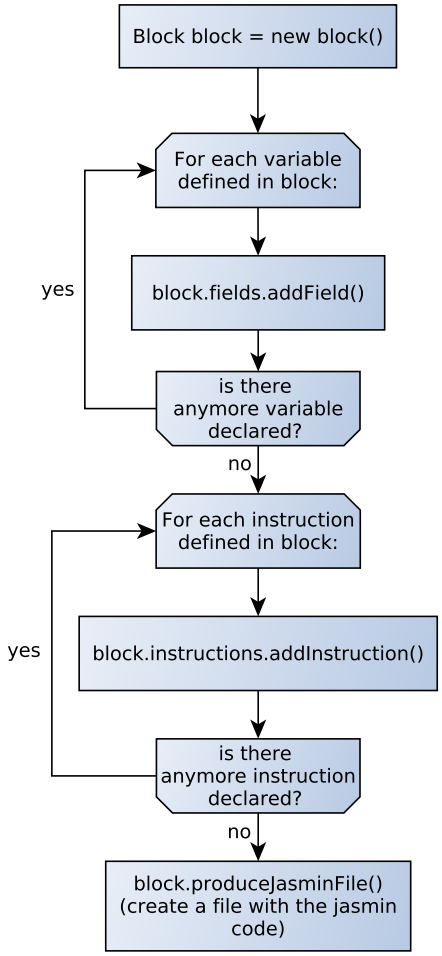

FIGURE 7 – Schéma général d'une production de code jasmin

Pour produire le code jasmin, la classe *Block* a comme attributs des classes spécialisées dans la "vraie" production du code jasmin : à savoir produire un texte jasmin en fonction de leur spécialité. Les attributs en question de la classe *Block* sont les suivants :

- constructor : de type ConstructorProducer
- localFields : de type Fields
- instructions : de type FunctionInstructions

ConstructorProducer s'occupe de créer le code jasmin concernant le constructeur d'une fonction ou du bloc main, *Field* s'occupe de créer la partie liée aux variables déclarées, et *FunctionInstruction* s'occupe de convertir les instructions du bloc en instructions jasmin. Chacune de ces classes crée son code jasmin dans une "string" de type *JasminExpression*. Cette string sera utilisée par *Block.produceJasminFile()* pour écrire le fichier jasmin.

4.4.3 JasminExpressionEvaluator

Enfin, pour convertir une classe Expression en du code jasmin, la classe *JasminExpressionEvaluator*. Je voulais regrouper toutes les fonctions pour convertir une expression en jasmin, afin d'éviter d'implémenter ces fonctions dans chaque classe Expression et gagner en lisibilité. Ce fut l'occasion de tester le *VisitorPattern* : *JasminExpressionEvaluator* implémente l'interface

permettant d'accéder à toutes les classes d'expression afin de les convertir en code jasmin.

5 Exemples de compilation / exécution

Afin d'illustrer le bon fonctionnement du compilateur, j'ai choisi les 4 cas suivants en essayant de regrouper la plus large palette de concepts définis dans le langage Hepial.

Le premier exemple ne produit pas de code, mais indique certaines erreurs de syntaxe principales à relever : l'affectation sur une constante, une expression entière dans une affectation booléenne, une double déclaration de variable, et l'utilisation d'une variable indéfinie.

Extrait de code 8 – Code Hepial pour tester les erreurs sémantiques

```
1 :   programme hepialProgramName
2 :
3 :   constante entier const = 1;
4 :   booleen bool;
5 :   entier bool;
6 :
7 :   debutprg
8 :      const = 123;
9 :      bool = bool + 321;
10 :     bool = indefini;
11 :  finprg
```

6 Améliorations possibles