Rapport du projet du module : Compilation

Semestre : 3

Option : Génie Logiciel

Sujet :

Réalisation d'un compilateur pour le langage R

Réalisé par :

GHAZLANE Mohammed (GL 2)

NACHIT Btissam (GL 3)

SABOUR Ilham (GL 3)

Sous la direction de :

Pr. TABII Youness

Année Universitaire : 2019 / 2020

Table des matières

Introduction

Un compilateur est un programme qui transforme un code source en un code objet. Généralement, le code source est écrit dans un langage de programmation (le *langage source*), il est de haut niveau d'abstraction, et facilement compréhensible par l'humain. Le code objet est généralement écrit en langage de plus bas niveau (appelé *langage cible*), par exemple un langage d'assemblage ou langage machine, afin de créer un programme exécutable par une machine.

Historiquement, Les logiciels des premiers ordinateurs étaient écrits en langage assembleur. Les langages de programmation de plus haut niveau (dans les couches d'abstraction) n'ont été inventés que lorsque les avantages apportés par la possibilité de réutiliser le logiciel sur différents types de processeurs sont devenus plus importants que le coût de l'écriture d'un compilateur. La capacité de mémoire très limitée des premiers ordinateurs a également posé plusieurs problèmes techniques dans le développement des compilateurs. Vers la fin des années 1950, des langages de programmation indépendants des machines font pour la première fois leur apparition. Par la suite, plusieurs compilateurs expérimentaux sont développés.

Dans ce contexte, nous allons développer un compilateur du langage R. On procédera d'abord par introduire ce langage, puis nous passerons à l'implémentation de l'analyseur lexical, puis nous aborderons l'analyseur syntaxique pour enfin générer l'analyseur sémantique.

Chapitre 1

Analyse du problème

1.1. Introduction au langage R :

1.1.1. Présentation du langage :

R est un langage de programmation et un logiciel libre destiné aux statistiques et à la science des données soutenu par la R Foundation for Statistical Computing. R fait partie de la liste des paquets GNU et est écrit en C (langage), Fortran et R.

GNU R est un logiciel libre distribué selon les termes de la licence GNU GPL et disponible sous GNU/Linux, FreeBSD, NetBSD, OpenBSD, Mac OS X et Windows.

Le langage R est largement utilisé par les statisticiens, les data miners, data scientists pour le développement de logiciels statistiques et l'analyse des données.

En janvier 2019, R est classé 12e dans l'index TIOBE qui mesure la popularité des langages de programmation.

1.1.2. Historique :

R est une implémentation du langage de programmation S avec l'ajout de la portée lexicale, inspirée du Scheme, et du ramasse-miettes (informatique).

Le langage S a été développé par John Chambers et ses collègues au sein des laboratoires Bell.

Le projet R naît en 1993 comme un projet de recherche de Ross Ihaka et Robert Gentleman à l'université d'Auckland (Nouvelle-Zélande).

Depuis 1997, une vingtaine de développeurs forment l'équipe de développement de R (R Development Core team). Les membres de cette équipe ont les droits d'écriture sur le code source. Le 23 avril 1997 débute le Comprehensive R Archive Network (CRAN) puis le 5 décembre 1997, R est intégré au Projet GNU.

La version R 1.0.0, première version officielle du langage R, est publiée le 29 février 2000.

En 2003, l'équipe de développement crée la R Foundation for Statistical Computing pour soutenir le projet R et devenir un point de contact de référence pour ceux qui veulent prendre contact avec la communauté R. À ce moment, le langage compte plus de 200 bibliothèques développées par la communauté scientifique qui utilise R.

La version 2.0 est publiée le 4 octobre 2004 et la version 3.0 le 3 avril 2013.

En 2015, plusieurs acteurs économiques importants comme IBM, Microsoft ou encore la société RStudio créent le R Consortium pour soutenir la communauté R et financer des projets autour de ce langage.

1.1.3. Lexique du langage R

Les caractéristiques principales du langage :

- ✓ R est un langage de programmation bien développé, simple et efficace qui comprend des conditions, des boucles, des fonctions récursives définies par l'utilisateur et des fonctionnalités d'entrée sortie.
- ✓ R dispose d'une installation efficace de traitement et de stockage des données,
- ✓ R fournit une suite d'opérateurs pour les calculs sur des tableaux, des listes, des vecteurs et des matrices.
- ✓ R fournit des fonctionnalités graphiques pour l'analyse des données et l'affichage soit directement sur l'ordinateur, soit sur papiers
- ✓ R permet la programmation procédurale et avec certaines fonctions la programmation orientée objet.

1.1.3.1 Les types d'objets dans R :

Dans tout langage informatique, les variables fournissent un moyen d'accéder aux données stockées en mémoire. R ne permet pas d'accéder directement à la mémoire de l'ordinateur mais fournit plutôt un certain nombre de structures spécialisées qui sont des objets. Ceux-ci sont désignés par des symboles ou des variables. Dans R, cependant, les symboles sont eux-mêmes des objets et peuvent être manipulés de la même manière que n'importe quel autre objet.

Les types basiques du langage R sont :

- ➢ **Vecteurs :**

Les vecteurs sont une structure de données de base dans R. Ils contiennent des éléments du même type. Les types de données peuvent être logiques, entiers, doubles, caractères, complexes ou bruts. Ils sont généralement créés à l'aide des fonctions prédéfinies **c()** ou **seq().**

- **Listes :**

Les listes ("vecteurs génériques") sont un autre type de stockage de données. Elles comportent des éléments qui ne sont pas nécessairement être du même type. Les éléments d'une liste sont accessibles par trois opérations d'indexation différentes. Les listes peuvent être créées grâce à la fonction prédéfinie **list() .**

- **Matrices :**

Les matrices sont une structure de données bidimensionnelle. Elles sont similaires aux vecteurs mais contiennent en plus l'attribut dimensionnel.

- **Tableaux de données :**

Les trames de données sont une structure de données bidimensionnelle dans R. C'est un cas particulier des listes dont chaque composante est de longueur égale. Chaque composante forme la colonne et le contenu de la composante forme les lignes.

- **Facteurs :**

Les facteurs sont une structure de données utilisée pour les champs qui ne prennent qu'un nombre fini de valeurs prédéfinies (données catégorielles). Par exemple : un champ de données tel que l'état civil ne peut contenir que des valeurs provenant de célibataires, mariés, séparés, divorcés ou veufs. Dans ce cas, nous connaissons à l'avance les valeurs possibles et ces valeurs prédéfinies et distinctes sont appelées des niveaux.

- **Fonctions :**

Dans R, les fonctions sont des objets et peuvent être manipulées à peu près de la même manière que n'importe quel autre objet. Elles ont trois composantes de base : une liste d'arguments formelle, un corps et un environnement. La liste d'arguments est une liste d'arguments séparés par des virgules. Un argument peut être un symbole, ou une construction "symbole = défaut", ou l'argument spécial "...". La deuxième forme d'argument est utilisée pour spécifier une valeur par défaut pour un argument. Cette valeur sera utilisée si la fonction est appelée sans qu'aucune valeur ne soit spécifiée pour cet argument. L'argument "..." est spécial et peut contenir un nombre quelconque d'arguments. Il est généralement utilisé si le nombre d'arguments est inconnu ou dans les cas où les arguments seront transmis à une autre fonction. Le corps est une

déclaration R analysée. Il s'agit généralement d'un ensemble d'énoncés entre accolades, mais il peut s'agir d'un seul énoncé, d'un symbole ou même d'une constante. L'environnement d'une fonction est l'environnement qui était actif lors de la création de la fonction. Pour déclarer une fonction on utilise le terme prédéfini function dont la structure est la suivante :

```
Nom_fonct <- function (argument) {
        Statement
}
```

- **Null :**

NULL est utilisé chaque fois qu'il est nécessaire d'indiquer ou de préciser qu'un objet est absent. Il ne doit pas être confondu avec un vecteur ou une liste de longueur zéro. L'objet NULL n'a pas de type et aucune propriété modifiable. Il n'y a qu'un seul objet NULL dans R, auquel toutes les instances se réfèrent.

1.1.4. La grammaire du langage :

La grammaire initialement définie est comme suit :

G = {T , NT ,S , P } avec :

T = {id, num, lettre, for, in, break, next, (,),",+,-,/,*,%%, %/%, function, if, else, return, repeat, by, length, while, =,!=,<> ,<,>,<=,>=, ^,<- }

Les règles de production sont :

INSTS → INST { DELIMITANT INSTS} | Ɛ

DELIMITANT → ; | Ɛ

INST → AFFEC | COMMANDE | FUNCTION | SI | POUR | TANTQUE | REPETER | EXPR

AFFEC → id AFFECT' AFFECT''

AFFECT' → = | <-

AFFECT'' → COMMANDE | EXPR

COMMANDE → id { PARAMETRE }

PARAMETRE → id {, PARAMETRE } | COMMANDE | Ɛ

FONCTION → function(ARGUMENT) { INSTS RETURN }

ARGUMENT → IDENTIFIANT ARGUMENT' | Ɛ

ARGUMENT' → , ARGUMENT | = EXPR { ,ARGUMENT] | Ɛ

IDENTIFIANT → id | CHAINE

CHAINE → ''letter ou chiffre ou alphanumerique''

SI → if (COND) { INSTS STOP } {{else if (COND) { INSTS STOP} else {INSTS}}

POUR → for (id in SEQ) {INSTS STOP }

REPETER → repeat { INSTS ; if (COND) break }

COND → EXPR VERF EXPR

EXPR → TERM OP TERM

TERM → id | num | (EXPR)

OP → + | - | * | / | ^ | %/% | %%

SEQ → id (PARAMETRE) | id | num : num | id(num , num , PAR = num)

VERF → <> | = | != | < | > | <= | >=

RETURN → return (INST) | Ɛ

PAR → by | length

STOP → break | next | Ɛ

1.2. Modification de la grammaire

La grammaire ci-dessus n'est pas LL(1) pour la rendre ainsi on procède par factorisation et élimination de la récursivité gauche. Les modifications apportées à la grammaire sont :

INST → id INST' | FONCTION | SI | POR | TANTQUE | REPETER | EXPR

INST' → AFFECT | COMMANDE | Ɛ

AFFECT'' → IDENTIFIANT AFFECT''' | FONCTION | EXPR

AFFECT ''' → COMMANDE | OP TERM

PARAMETRE → id PARAMETRE' | Ɛ

PARAMETRE' → , PARAMETRE | COMMANDE

1.3. Grammaire LL(1):

La grammaire LL(1) résultante après la modification est :

INSTS → INST { DELIMITANT INSTS} | Ɛ

DELIMITANT → ; | Ɛ

INST → id INST' | FONCTION | SI | POR | TANTQUE | REPETER | EXPR

INST' → AFFECT | COMMANDE | Ɛ

AFFEC → id AFFECT' AFFECT''

AFFECT' → = | <-

AFFECT'' → IDENTIFIANT AFFECT''' | FONCTION | EXPR

AFFECT ''' → COMMANDE | OP TERM

COMMANDE → id { PARAMETRE }

FONCTION → function(ARGUMENT) { INSTS RETURN }

PARAMETRE → id PARAMETRE' | Ɛ

PARAMETRE' → , PARAMETRE | COMMANDE

ARGUMENT → IDENTIFIANT ARGUMENT' | Ɛ

ARGUMENT' → , ARGUMENT | = EXPR { ,ARGUMENT] | Ɛ

IDENTIFIANT → id | CHAINE

CHAINE → ''letter ou chiffre ou alphanumerique''

SI → if (COND) { INSTS STOP } {{else if (COND) { INSTS STOP} else {INSTS}}

POUR → for (id in SEQ) {INSTS STOP }

REPETER → repeat { INSTS ; if (COND) break }

COND → EXPR VERF EXPR

EXPR → TERM OP TERM

TERM → id | num | (EXPR)

OP → + | - | * | / | ^ | %/% | %%

SEQ → id (PARAMETRE) | id | num : num | id(num , num , PAR = num)

VERF → <> | = | != | < | > | <= | >=

RETURN → return (INST) | Ɛ

PAR → by | length

STOP → break | next | Ɛ

Chapitre 2

Réalisation du compilateur

2.1. Analyseur lexical

Tout d'abord, nous commencerons par présenter les tokens des mots clés et ceux des caractères spéciaux du langage et leurs significations.

2.1.1. Les mots clés

Mot clé	*Nom du token*
If	IF_TOKEN
Else	ELSE_TOKEN
Repeat	REPEAT_TOKEN
While	WHILE_TOKEN
Function	FUNCTION_TOKEN
Return	RETURN_TOKEN
null	NULL_TOKEN
For	FOR_TOKEN
In	IN_TOKEN
Next	NEXT_TOKEN
Break	BREAK_TOKEN
True	TRUE_TOKEN
false	FALSE_TOKEN

Parmi ces mots, if, else, repeat, while, function, for, in, next et break sont utilisés pour les conditions, les boucles et les fonctions définies par l'utilisateur.

True et false sont des opérateurs logiques, function est le mot réservé pour la définition d'une fonction par l'utilisateur.

2.1.2. Les opérateurs

- **Les opérateurs arithmétiques :**

Opérateur	*Description*	*Nom du token*
+	Addition	PLUS_TOKEN
-	Soustraction	MOINS_TOKEN
*****	Multiplication	MULT_TOKEN
/	Division	DIV_TOKEN
^	Puissance	PUISS_TOKEN
%%	Modulo	REST_TOKEN
%/%	Division entière	ENTIER_TOKEN

- **Les opérateurs relationnels :**

Opérateur	*Description*	*Nom du token*
<	Inférieur	INF_TOKEN
>	Supérieur	SUP_TOKEN
<=	Inférieur ou égal	INFEG_TOKEN
>=	Supérieur ou égal	SUPEG_TOKEN
==	Egal	EGAL_TOKEN
!=	Différent	DIFF_TOKEN

2.1.3. Les caractères spéciaux

Caractère	*Nom du token*
,	VIR_TOKEN
.	PT_TOKEN
;	PV_TOKEN
:	DPT_TOKEN
(	PO_TOKEN
)	PF_TOKEN
{	ACCO_TOKEN
}	ACCF_TOKEN
[	CRO_TOOKEN
]	CRF_TOKEN
\n	RETOUR_TOKEN
``	APS_TOKEN
= , <-	AFF_TOKEN
EOF	EOF_TOKEN

2.1.4. Les règles lexicales

ID	ID_TOKEN
Numéro	NUM_TOKEN
Le reste	ERREUR_TOKEN

Les fonctions prédéfinies dans le langage sont déclarées en tant que identificateur (ID_TOKEN) et seront définies dans l'analyseur syntaxique.

Les métarègles qui définissent la forme d'un programme en langage R sont :

- Un commentaire est une suite de caractères commençant par un #
- Un séparateur est un retour à la ligne '\n', une tabulation '\t' , un retour chariot '\r' ou un commentaire

- Deux ID ou mots clés qui se suivent doivent être séparés par au moins un séparateur
- Des séparateurs peuvent être insérés partout sauf à l'intérieur d'un terminal
- Un identificateur ne doit pas commencer par un chiffre

2.1.5. Les erreurs

Tout caractère non reconnu par le compilateur et qui ne fait pas partie des catégories précédentes est déclaré comme erreur qui prend comme token : ERREUR_TOKEN et affiche un message d'erreur. Les erreurs définies sont :

Erreur	Message
ERR_CAR_INC	Caractère inconnu
ERR_FICH_VID	Fichier vide
ERR_COM	Erreur commentaire
ERR_STRING	Chaine de caractères inconnue

2.1.6. Fonctions de l'analyseur lexical

On distingue cinq catégories de symboles : les mots, les nombres, les chaînes, les symboles spéciaux et les erreurs. Les fonctions suivantes lisent ces différentes catégories et indiquent leur nature :

- ***void Lire_Caractere() :*** c'est la fonction qui parcourt le fichier qui contient le code caractère par caractère.
- ***void Lire_Mots() :*** la fonction qui lit un mot et précise s'il est un simple identifiant ou s'il est un mot clé.
- ***void Lire_Nombre() :*** la fonction qui lit un nombre.
- ***void Lire_Special() :*** la fonction qui lit les caractères spéciaux et les opérateurs
- ***void Lire_Commentaire() :*** la fonction qui lit les commentaires mais ne les déclare pas
- ***void Lire_String() :*** la fonction qui lit les chaîne de caractère délimitées par des apostrophes et n'affiche pas leurs contenus.
- ***void Sym_Suiv() :*** la fonction qui lit un seul caractère et appelle la fonction concernée parmi les précédentes.
- ***void Erreur(Erreurs rerr) :*** la fonction qui affiche à l'écran l'erreur rencontrée et son message associé.

- ***void AfficherToken(TSym_Cour sc) :*** la fonction qui affiche le caractère lu et son token.

2.1.7. Organisation du code

Notre compilateur est composé d'un header « Analyseur_lexical.h » , d'un fichier c « Analyseur_lexical.c » et d'un fichier de test « main.c ».

- **Le header contient :**

⇨ Tous les tokens déclarés ci-dessus regroupés dans une énumération nommée CODE_LEX, chacun étant identifié par un entier.

```
typedef enum {
ID_TOKEN, IF_TOKEN, ELSE_TOKEN, WHILE_TOKEN, REPEAT_TOKEN, BREAK_TOKEN,NEXT_TOK
EN,
FOR_TOKEN, IN_TOKEN, FUNCTION_TOKEN, RETURN_TOKEN, TRUE_TOKEN, FALSE_TOKEN,NULL
_TOKEN, PV_TOKEN, PT_TOKEN, PLUS_TOKEN,MOINS_TOKEN, MULT_TOKEN, DIV_TOKEN, VIR_
TOKEN,
AFF_TOKEN, INF_TOKEN, INFEG_TOKEN, SUP_TOKEN, SUPEG_TOKEN, EGAL_TOKEN, DIFF_TOK
EN, PO_TOKEN, PF_TOKEN, ACCO_TOKEN, ACCF_TOKEN, CRO_TOKEN, CRF_TOKEN, APS_TOKEN
,
 NUM_TOKEN, ERREUR_TOKEN, COM_TOKEN, DPT_TOKEN,PUISS_TOKEN, REST_TOKEN,ENTIER_T
OKEN, EOF_TOKEN, RETOUR_TOKEN, STRING_TOKEN
} CODE_LEX ;
```

⇨ Les erreurs sont déclarées dans une énumération nommée Erreurs.

```
typedef enum {
    ERR_CAR_INC, ERR_FICH_VID, ERR_COM, ERR_STRING
}Erreurs;
```

⇨ On définit une structure TSym_Cour qui contient deux champs code et nom qui sont respectivement le token et le caractère associé.

```
typedef  struct { CODE_LEX CODE; char NOM[20]; } TSym_Cour;
```

⇨ Les prototypes de toutes les fonctions vues précédemment

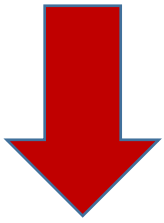
- **Le fichier c « Analyseur_lexical.c » :** contient les définitions des fonctions et les traitements qu'elles font.
- **Le fichier main :** contient le code qui teste le programme

2.1.8. Exécution du code

Ci-dessous on retrouve le résultat de l'exécution de notre analyseur lexical sur un exemple de programme écrit en R.

```
v <- LETTERS[1:16];
for ( i in v) {

if (i == "D") {
next
}
print(i)
}
pow <- function(x, y) {
# function to print x raised to the power y
result <- x^y
print(paste(x,"raised to the power", y, "is", result))
}
IF (x > 0) {
result <- "Positive"
}
else if (x < 0) {
result <- "Negative"
}
return(result)
}
r=1%/%2
m= 2%%4
```



v : ID_TOKEN

<- : AFF_TOKEN

LETTERS : ID_TOKEN

[: CRO_TOKEN

1 : NUM_TOKEN

: : DPT_TOKEN

16 : NUM_TOKEN

] : CRF_TOKEN

; : PV_TOKEN

: RETOUR_TOKEN

for : FOR_TOKEN

(: PO_TOKEN

i : ID_TOKEN

in : IN_TOKEN

v : ID_TOKEN

) : PF_TOKEN

```
{   : ACCO_TOKEN
    : RETOUR_TOKEN
    : RETOUR_TOKEN
 if   : IF_TOKEN
 (   : PO_TOKEN
 i   : ID_TOKEN
 ==   : EGAL_TOKEN
 "   : STRING_TOKEN
 )   : PF_TOKEN
 {   : ACCO_TOKEN
    : RETOUR_TOKEN
 next   : NEXT_TOKEN
    : RETOUR_TOKEN
 }   : ACCF_TOKEN
    : RETOUR_TOKEN
 print   : ID_TOKEN
 (   : PO_TOKEN
 i   : ID_TOKEN
 )   : PF_TOKEN
    : RETOUR_TOKEN
 }   : ACCF_TOKEN
   : RETOUR_TOKEN
 pow   : ID_TOKEN
 <-   : AFF_TOKEN
 function   : FUNCTION_TOKEN
 (   : PO_TOKEN
 x   : ID_TOKEN
 ,   : VIR_TOKEN
 y   : ID_TOKEN
 )   : PF_TOKEN
 {   : ACCO_TOKEN
    : RETOUR_TOKEN
```

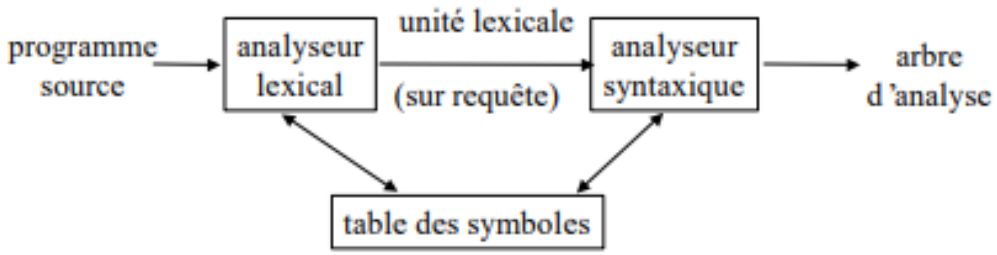
```
: RETOUR_TOKEN
 result   : ID_TOKEN
 <-   : AFF_TOKEN
 x   : ID_TOKEN
 ^   : PUISS_TOKEN
 y   : ID_TOKEN
    : RETOUR_TOKEN
 print   : ID_TOKEN
 (   : PO_TOKEN
 paste   : ID_TOKEN
 (   : PO_TOKEN
 x   : ID_TOKEN
 ,   : VIR_TOKEN
 "   : STRING_TOKEN
 ,   : VIR_TOKEN
 y   : ID_TOKEN
 ,   : VIR_TOKEN
 "   : STRING_TOKEN
 ,   : VIR_TOKEN
 result   : ID_TOKEN
 )   : PF_TOKEN
 )   : PF_TOKEN
    : RETOUR_TOKEN
 }   : ACCF_TOKEN
    : RETOUR_TOKEN
IF   : ID_TOKEN
 (   : PO_TOKEN
 x   : ID_TOKEN
 >   : SUP_TOKEN
 0   : NUM_TOKEN
 )   : PF_TOKEN
 {   : ACCO_TOKEN
```

```
   : RETOUR_TOKEN
result   : ID_TOKEN
<-   : AFF_TOKEN
"   : STRING_TOKEN
   : RETOUR_TOKEN
}   : ACCF_TOKEN
   : RETOUR_TOKEN
else   : ELSE_TOKEN
if   : IF_TOKEN
(   : PO_TOKEN
x   : ID_TOKEN
<   : INF_TOKEN
0   : NUM_TOKEN
)   : PF_TOKEN
{   : ACCO_TOKEN
   : RETOUR_TOKEN
result   : ID_TOKEN
<-   : AFF_TOKEN
"   : STRING_TOKEN
   : RETOUR_TOKEN
```

```
}   : ACCF_TOKEN
   : RETOUR_TOKEN
return   : RETURN_TOKEN
(   : PO_TOKEN
result   : ID_TOKEN
)   : PF_TOKEN
   : RETOUR_TOKEN
}   : ACCF_TOKEN
 : RETOUR_TOKEN
r   : ID_TOKEN
=   : AFF_TOKEN
%/%   : ENTIER_TOKEN
2   : NUM_TOKEN
   : RETOUR_TOKEN
m   : ID_TOKEN
=   : AFF_TOKEN
2   : NUM_TOKEN
%%   : REST_TOKEN
4   : NUM_TOKEN
```

2.2. Analyseur syntaxique

2.2.1. Rôle de l'analyseur syntaxique



Rôle central de la partie frontale :

- Active l'analyseur lexical.
- Vérifie la conformité syntaxique.
- Construit l'arbre d'analyse.
- Prépare ou anticipe la traduction.
- Gère les erreurs communes de syntaxe.

2.2.2. Fonctions de l'analyseur syntaxique

L'analyseur syntaxique regroupe l'ensemble des procédures récursives, chaque procédure correspond à une règle syntaxique, qui est elle-même une règle de production.

Les fonctions définies sont les suivantes :

- **void INSTS() ;**
- **void INST();**
- **void INST_PRIME();**
- **void AFFECT();**
- **void AFFECT_PRIME();**
- **void AFFECT_PRIME_SECOND();**
- **void AFFECT_PRIME_TROIS();**
- **void FUNCTION();**
- **void ARGUMENT();**
- **void ARGUMENT_PRIME();**
- **void COMMANDE();**
- **void PARAMETRE();**
- **void PARAMETRE_PRIME();**
- **void SI();**

- **void POUR();**
- **void TANTQUE();**
- **void REPETER();**
- **void COND();**
- **void EXPR();**
- **void TERM();**
- **void OP();**
- **void VERF();**
- **void RETURN();**
- **void STOP();**
- **void SEQ();**
- **void Premier_sym();**
- **void IDENT();**

2.2.3. Organisation du code

Notre compilateur est composé d'un header « Analyseur_syntaxique.h » , d'un fichier c « Analyseur_syntaxique.c » et d'un fichier de test « main.c ».

- **Le header contient :** Toutes la liste des prototypes des fonctions présentées ci-dessus.
- **Le fichier c « Analyseur_syntaxique.c » :** contient les définitions des fonctions qui ont déjà déclaré sur le header et les traitements qu'elles font.
- **Le fichier main :** contient le code qui teste le programme.

2.2.4. Exécution du code

Ci-dessous on retrouve le résultat de l'exécution de notre analyseur syntaxique sur un exemple de programme écrit en R.

```
getwd()
setwd("D:/1-
Ensias S3/M5.Statistiques et analyse de données/analyse donnee/TP/tp2")
getwd()
a = 1
v = 5
for ( i in v ) {
  if (i == a) {
    a = 3
  }
}
```

```r
x = 1
y = 2+x
pow <- function(x, y) {
  # function to print x raised to the power y
  result <- x^y
  print(x,"raised to the power", y, "is", result)
  matrix(1,20,30)
  help("print")
}

check <- function(x) {
  if (x > 0) {
    result <- "Positive"
  }
  else {
    result <- "Zero"
  }
  return(result)
}

1%/%2
2%%4

x <- 1

repeat {
  print(x)
  x = x+1
  if (x == 6){
    break
  }
}

i <- 1
while (i < 6) {
  print(i)
  i = i+1
}
```

⇨ Résultat éxecution :

- Cas succès :

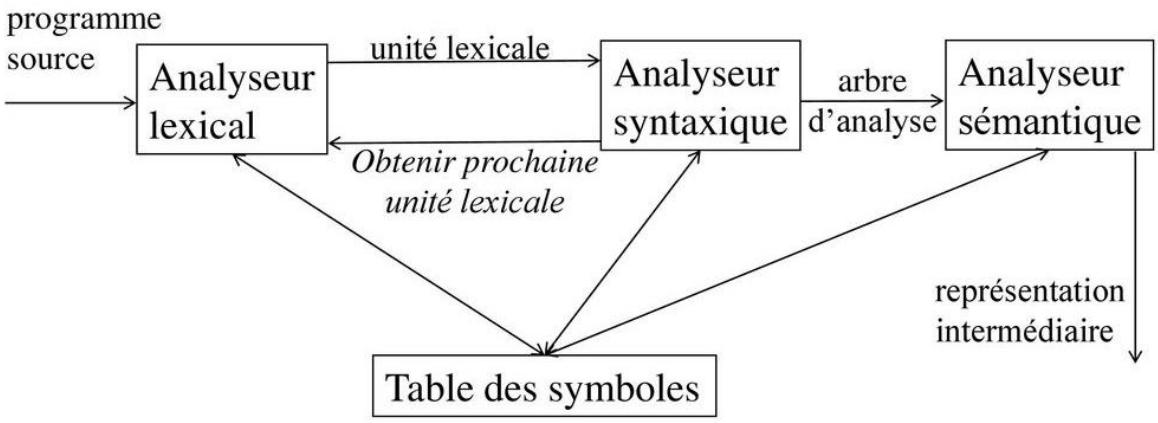
```
**************BRAVO: le programme est correcte!!!**************
```

- Cas échec :

```
Erreur  numero  0        : caractere inconnu  video
```

2.3. Analyseur sémantique

2.3.1. Rôle de l'analyse sémantique



L'analyse sémantique est une technique proche de l'analyse lexicale, mais au lieu de se faire au niveau des mots, l'analyse se fait sur le sens des phrases pour déterminer le sens des écrits, et c'est ce qui en fait toute la différence.

L'analyseur sémantique consiste à étudier le sens des mots dans leur contexte. Ainsi d'établir la signification d'une phrase en utilisant le sens des éléments la composant.

2.3.2. Règles sémantiques

La liste suivante représente l'ensemble des règles sémantique traitées sur notre compilateur :

- Tous les identifiants utilisés dans les affectations, les commandes …. Doivent être déjà définies
- Les arguments d'une fonction ne peuvent être utilisée que dedans cette fonction.
- Un id d'une fonction ne peut être redéfini d'une autre manière
- Les id des commandes doivent être déjà existées
- Les instances dans une seule ligne doivent être séparées par des ;
- Les fonctions utilisées dans un block doivent être déclarer ailleurs.

- Pour les fonctions et les commandes il faut respecter le nombre de paramètre au moment de l'appel

2.3.3. Fonctions de l'analyseur sémantique

On distingue sept règles de gestion qui ont présentés au-dessus. Les fonctions suivantes traitent ses différentes catégories et indiquent leur validité :

- **int is_declared(char * nom) :** Cette fonction vérifier si une variable est déjà déclarée ou non.
- **int is_fonction(char * nom) :** Cette fonction permet de vérifier si le nom passer en paramètre représente une fonction déjà définie.
- **int is_nb_parameter(char * nom,int nb) :** La fonction qui fait une comparaison entre le nombre de paramètre défini au moment de la déclaration de la commande et le nombre de paramètre utilisés au moment de l'appel de la commande.
- **int is_parameter(char * nom) :** Fonction qui permet de vérifier si le nom du paramètre envoyé à la commande est valide.
- **int is_argument(char * nom) :** Fonction qui permet de vérifier si le nom de l'argument passer en paramètre est valide.
- **int is_command(char * nom) :** Fonction qui permet de vérifier si le nom de la commande passer en paramètre est valide.
- **int is_nb_argument(char * nom,int nb) :** La fonction qui fait une comparaison entre le nombre d'argument défini au moment de la déclaration de la fonction et le nombre de paramètre utilisés au moment de l'appel de la fonction.

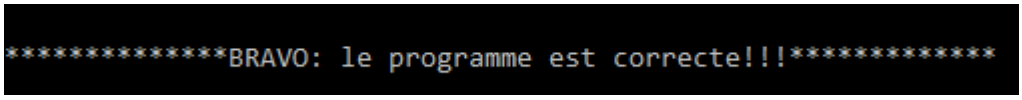
2.3.4. Organisation du code

Notre compilateur est composé d'un header « Analyseur_semantique.h » , d'un fichier c « Analyseur_semantique.c » et d'un fichier de test « main.c ».

- **Le header contient :** Toutes la liste des prototypes des fonctions présentées ci-dessus.
- **Le fichier c « Analyseur_semantique.c » :** contient les définitions des fonctions qui ont déjà déclaré sur le header et les traitements qu'elles font.
- **Le fichier main :** contient le code qui teste le programme.

2.3.5. Exécution du code

Pour le test on autilisé le même code présenté sur la partie de l'analyse syntaxique, et à la fin de l'exécution le programme nous donne le résultat suivant :



Conclusion

Ce projet nous permis de mettre en œuvre les concepts élémentaires de compilation de langage de programmation moderne.

L'expérience était très intéressante, et nous a donné l'occasion de découvrir énormément de problèmes liés notamment à la conception de la grammaire du langage R, qui n'étaient pas aussi évidents durant la partie de développement et de réalisation de code.

Malgré ces difficultés, le résultat est donc un compilateur qui semble celui de langage R et qui est prêt, efficace et pensé sans ambiguïté qui respecte la chaine de compilation d'un code : analyseur lexical, syntaxique et sémantique.

Comme perspectives on peut envisager par la suite à ajouter de nouvelles fonctionnalités comme l'amélioration de notre grammaire, l'ajout des nouveaux règles sémantiques et l'ajout d'un générateur de code.