Ecole Nationale Supérieure d'Informatique et d'Analyse des Systèmes - RABAT

Rapport de projet de Compilation

Mini compilateur en C

Filière : Ingénierie des systèmes embarqués et mobiles (ISEM)

Réalisé par :

Abdessalam Benayyad
AnassDkheila
Omar Dahmouni
Yasser Kharj
Najlae Karara

Encadré par :

M. Youness TABII

Année académique 2021/2022

Remerciements...

Nous tenons tout d'abord à remercier DIEU le tout puissant, qui nous a donné la force et la patience d'accomplir ce Modeste travail.

Nous tenons aussi à exprimer notre profonde reconnaissance à toutes les personnes qui ont contribuées,par leur aide et assistance,à la réalisation et au bon déroulement de notre projet.Ainsi, nous exprimons notre profonde gratitude et nos sincères remerciements à notre encadrant **Pr.Taabi** sa directive précieuse et ses conseils pertinents. Nous adressons nos remerciements également aux jurys, qui nous ont honorés en acceptant d'évaluer notre travail. De façon plus générale, nous tenons à adresser nos plus sincères remerciements à l'ensemble du corps enseignant de l'ENSIAS, d'avoir porté un vif intérêt à notre formation, et d'avoir accordé le plus clair de leur temps, leur attention et leur énergie, et ce, dans un cadre généreusement agréable.

Introduction

Ce projet représente notre travaille sur la réalisation d'un compilateur pour le **langage R** a base du **langage C**. Et nous avons travaillé avec le langage C et sans l'utilisation des générateurs de l'analyseur lexicale comme **FLEX** ou les générateurs de l'analyseur syntaxique comme **BISON**. Les trois éléments essentiels d'un compilateur sont :

- **l'analyseur lexicale**
- **l'analyseur syntaxique**
- **L'analyseur sémantique**

Nous nous sommes concentrés premièrement dans l'élaboration de la grammaire du langage pour faciliter la phase de codage. Ensuite nous avons diviser les tâches entre nous. La fin de chaque éléments du compilateur a été testé indépendamment des autres pour s'assurer de son bon fonctionnement premièrement. Après les avoir liés on les a testé sur différents instructions que notre compilateur doit reconnaître sans erreur et qui sont contenus dans le fichier **test_1.txt** qui est inclus dans le code envoye.

Le plan de ce rapport sera le suivant :

- **I. Grammaire**
- **II. Documentation :**
- **1- Analyseur Lexicale**
- **2- Analyseur Syntaxique**
- **3- Analyseur Sémantique**
- **III. Fonctionnement du compilateur**
- **IV. Conclusion**

I. Grammaire :

Nous avons écrit notre grammaire sans la rendre **LL(1)** pour qu'elle soit lisible, mais le codage de cette grammaire respecte tous les règles d'une grammaire **LL(1)**.

- **Prog :** [expr | Loop | Function | Decision]*
- **expr :** id <- (id | CallFunction | data) [operator (id |
- CallFunction | data)]* | id <- function | (id | CallFunction | data) [operator (id | CallFunction | data)]* -> id
- **operation :** (id | CallFunction | data) operator (id | CallFunction | data)
- **Operator :** == | < | > | + | - | * | /
- **AssignOp :** <- | = | ->
- **data :** INT | FLOAT | STRING | VECTOR
- **Decision :** if (condition) expr | if (condition) expr else expr
- **Loop :** while (condition) expr | repeat expr | for (ID in vector) expr
- **vector :** c(id [, id]*)
- **Function :** function (None | id [, id]*) expr | return
- CallFunc id (None | data [,data]*)

II. Documentation :

Analyseur Lexical

- **char Char_suivant()**
  - Description : Lit le prochain charactère et le retourne.
  - Arguments : NULL
- **void Erreur**$_{S}yntax(char * ErrorText)$
  - Description : Affiche l'erreur syntaxique rencontré et termine l'exécution du programme.
  - Arguments : ErrorText : le texte à afficher.

void Token_suivant()

- Description : Après avoir ignorer les séparateurs, elle teste si le prochain Token est un Nombre, Mot réservé (pour les boucles, les décisions, les fonctions) ou un charactère spécial. Sinon elle le traite comme un invalide Token, et arrête le programme.
- Arguments : NULL

void ignorer_Separateurs()

- Description : Ignore les commentaires ainsi que les séparateurs comme la tabulation, les espaces et les nouvelles lignes.
- Arguments : NULL

void est_Nombre()

- Description : Construit le nombre et vérifie si c'est un entier ou un réel. Sinon elle le traite comme un invalide Nombre, et arrête le programme.

- Arguments : NULL

void est_mot()

- Description : Construit le mot, et vérifie si c'est un mot réservé pour la fonction, les décisions, les boucles ou pour le vecteur. Sinon elle test si c'est un appel de fonction où le considère comme un identificateur.
- Arguments : NULL

int Operateur()

- Description : Vérifie si c'est opérateur tel la division, la multiplication, la soustraction, l'addition.
- Arguments : NULL

int est_Special()

- Description : Vérifie si c'est un charactère spécial, et en cas ou elle trouve les guillemets, elle vérifie si c'est un texte.
- Arguments : NULL

Analyseur Syntaxique :

- **int TestToken(TOKEN**$_C ODE Token To Test$)

  - Description : Teste si le Token de l'argument convient avec le Token actuel trouvé par l'analyseur lexicale. En cas de succès il fait appel au prochain Token.
  - Arguments :
  - TokenToTest : représente le Token qu'on va comparer avec le Token de l'analyseur lexicale.

void fin_Expr()

- Description : Assure qu'il y a un saut de ligne ou un point-virgule à la fin d'une expression. Autre symbole n'est pas permis, et donnera une erreur syntaxique.
- Arguments : NULL

void NoNewline()

- Description : Assure qu'il n'y aura pas un saut de ligne. En cas de trouvaille d'un saut de ligne, il y aura une erreur syntaxique.
- Arguments : NULL

int est_Expr()

- Description : Vérifie si la syntaxe représentée dans la grammaire de l'expression est valide. Voir la grammaire de : « expr » ;
- Arguments : NULL

int est_Decision()

- Description : Teste si l'expression actuel représente une instruction conditionnelle sous la forme de : if (condition) expressions ou bien if (condition) expressions else expressions
- Arguments : NULL

int est_Fonction()

- Description : Teste si l'expression actuel représente une déclaration de fonction sous la forme : function (arguments) expressions
- Arguments : NULL

int est_Retourner()

- Description : Teste si l'expression actuel est une instruction de retour pour les fonctions.
- Arguments : NULL

int est_boucle()

- Description : Puisqu'il y a trois instructions de boucle dans R qui sont : for, while, repeat. Alors cette fonction teste laquelle convient avec l'expression actuel.
- Arguments : NULL

int est_Vecteur()

- Description : Teste si l'expression actuel représente un vecteur.
- Arguments : NULL

int est_ID()

- Description : Teste si le Token donné par l'analyseur lexicale est un identificateur.
- Arguments : NULL

int Assigner_Op(int assignop)

- Description : Teste si le Token donné par l'analyseur lexicale représente le type d'affectation donné comme argument à cette fonction.
- Arguments : NULL

int est_Condition()

- Description : Teste si l'expression actuel représente la syntaxe d'une instruction conditionnelle qui est sous cette forme : un id ou appel fonction ou donnée plus un operateur de

comparaison plus un id ou appel fonction ou donnée.

- Arguments : NULL

int ID_Appfct_Donnee_Operation()

- Description : Teste si l'expression actuel représente la syntaxe d'une instruction conditionnelle qui est sous cette forme : un id ou appel fonction ou donnée plus un operateur de comparaison plus un id ou appel fonction ou donnée.
- Arguments : NULL

int est_Operateur(int type)

- Description : Vérifie si le Token est un opérateur normal comme la multiplication, addition, division, soustraction ou un opérateur de comparaison comme inférieur, inférieur ou égal ....
- Arguments :
- Type : type d'opérateur qu'on veut tester, normal ou de condition.

int Appel_fct()

- Description : Teste si l'expression d'appel fonction avec ses arguments est valide.
- Arguments : NULL

int est_Donnee()

- Description : Vérifie si le Token actuel représente une donnée de type : entier, réel, chaine de charactères ou un vecteur.
- Arguments : NULL

Analyseur Smantique :

void SemanticID(Symbol Input, Symbol Variable)

- Description : S'assure que l'identificateur reçu dans l'argument d'input est déjà déclaré afin qu'elle transfère son type ainsi que sa valeur à l'identificateur reçu dans l'argument de Variable.
- Arguments :
- Input : L'identificateur qui va être affecté à la variable.
- Variable : L'identificateur qu'on va lui affecter la valeur ainsi que le type d'Input.

void SemanticData(Symbol Input, Symbol Variable)

- Description : Attribue la valeur de la donnée reçue dans l'argument Input ainsi que son type à la variable reçu dans l'argument Variable.
- Arguments :
- Input : La donnée qu'elle va être affecté à la variable.
- Variable : L'identificateur qu'on va lui affecter la valeur ainsi que le type d'Input.

int ID$_{D}eclarer(Symboltoken)$

Description : S'assure que l'identificateur reçu dans l'argument Token est déjà déclaré. Dans ce cas elle retourne sa position dans la table de symbole, sinon elle retourne la valeur UNDECLARED.

Arguments :

Token : L'identificateur qu'on veut s'assurer s'il est déclaré ou non.

void printSymbolTable()

- Description : Affiche toute la table de symbole.
- Arguments : NULL

III.Fonctionnement du compilateur

```
|_____________________________ DÉBUT DE COMPILATION _____________________________|

ID  LEFT_ASSIGN INT
ID  LEFT_ASSIGN INT
ID  RIGHT_ASSIGN    ID
ID  LEFT_ASSIGN STRING  PLUS_OP STRING
ID  LEFT_ASSIGN INT
INT RIGHT_ASSIGN    ID
ID  LEFT_ASSIGN OPEN_PAR    INT CLOSE_PAR   MINUS_OP    INT
ID  LEFT_ASSIGN OPEN_PAR    OPEN_PAR    OPEN_PAR    OPEN_PAR    OPEN_PAR    INT CLOSE_PAR   CLOSE_PAR   CLOSE_PAR   CLOSE_PAR   MULT_OP INT CLOSE_PAR   DIV_OP  INT
ID  LEFT_ASSIGN STRING
ID  LEFT_ASSIGN INT
VECTOR  OPEN_PAR    FLOAT   COLON   STRING  CLOSE_PAR   RIGHT_ASSIGN    ID
ID  LEFT_ASSIGN STRING
ID  LEFT_ASSIGN CALLFUNC    OPEN_PAR    ID  COLON   ID  CLOSE_PAR
ID  LEFT_ASSIGN ID  MULT_OP ID  PLUS_OP ID
STRING  RIGHT_ASSIGN    ID
CALLFUNC    OPEN_PAR    INT COLON   INT COLON   FLOAT   CLOSE_PAR   RIGHT_ASSIGN    ID
FUNCTION    OPEN_PAR    CLOSE_PAR   OPEN_BRACE  CLOSE_BRACE RIGHT_ASSIGN    ID
FUNCTION    OPEN_PAR    ID  COLON   ID  COLON   ID  CLOSE_PAR   OPEN_BRACE  CLOSE_BRACE
ID  LEFT_ASSIGN FUNCTION    OPEN_PAR    CLOSE_PAR   OPEN_BRACE
ID  LEFT_ASSIGN INT SEMICOLON
RETURN  INT SEMICOLON
CLOSE_BRACE
CALLFUNC    OPEN_PAR    ID  COLON   ID  COLON   ID  CLOSE_PAR   RIGHT_ASSIGN    ID
CALLFUNC    OPEN_PAR    STRING  COLON   INT COLON   INT CLOSE_PAR
CALLFUNC    OPEN_PAR    INT CLOSE_PAR   RIGHT_ASSIGN    ID
IF  OPEN_PAR    ID  SUP_OP  ID  CLOSE_PAR   OPEN_BRACE
ID  LEFT_ASSIGN INT
CLOSE_BRACE
ELSE    OPEN_BRACE
ID  LEFT_ASSIGN INT
CLOSE_BRACE
FOR OPEN_PAR    ID  IN  VECTOR  OPEN_PAR    INT COLON   STRING  COLON   FLOAT   CLOSE_PAR   CLOSE_PAR   OPEN_BRACE
INT RIGHT_ASSIGN    ID
CLOSE_BRACE
REPEAT  OPEN_BRACE
STRING  RIGHT_ASSIGN    ID
CLOSE_BRACE
WHILE   OPEN_PAR    ID  INF_OP  ID  CLOSE_PAR   OPEN_BRACE
ID  LEFT_ASSIGN STRING
CLOSE_BRACE
```

```
Nom  :e     |||| Valeur  :10    |||| Type  :INT

Nom  :a     |||| Valeur  :2     |||| Type  :INT

Nom  :b     |||| Valeur  :5     |||| Type  :INT

Nom  :zz    |||| Valeur  :3+2   |||| Type  :STRING

Nom  :c     |||| Valeur  :(((((4))))*2)/5   |||| Type  :INT

Nom  :Nom   |||| Valeur  :Yasser    |||| Type  :STRING

Nom  :Age   |||| Valeur  :21    |||| Type  :INT

Nom  :Vecteur   |||| Valeur  :)     |||| Type  :CLOSE_PAR

Nom  :Filiere   |||| Valeur  :ISEM  |||| Type  :STRING

Nom  :result    |||| Valeur  :(25)-88*25+(((((4))))*2)/5    |||| Type  :INT

Nom  :string    |||| Valeur  : Bonjour      |||| Type  :STRING

Nom  :k     |||| Valeur  :10    |||| Type  :INT

Nom  :run   |||| Valeur  :TEST  |||| Type  :STRING

Nom  :ab    |||| Valeur  :deux  |||| Type  :STRING


|_____________________________ FIN DE COMPILATION _____________________________|
```

```r
# C'est un commentaire

# Differentes formes d'expressions

e <- 10

a <- 1

a -> b

zz <- "3" + "2"

c <- 10

25 -> b

a <- ( 25 ) - 88

c <- (((((4))))*2)/5

Nom <- "Yasser"

Age <- 21

c(222.22 ,"bnj") -> Vecteur

Filiere <- "ISEM"

Moyenne <- calculer_Moy(x,y)

result <- a * b + c

" Bonjour " -> string

Matrix(3,2,3.5) -> a


```

```
# Differentes formes de fonctions

function(){} -> b

function(x,y ,z ){}

k <- function(){


b <- 5;
return 0;
}


# Appel  de fonction


n(x,y,z) -> fonction_n

hello("Mardi",12,2022)

appel(212) -> test


# Condition

if( alpha > beta){
    k <- 5
}
else{
    k <- 10
}


# Différents formes de boucles


for(a in c(2,"string",3.14)){
    2 -> a
}

repeat{
   "TEST" -> run
}

while( a<b){
    ab <- "deux"
}
```

IV.Conclusion

Ce projet nous a permis de parfaitement comprendre la chaîne de compilation d'un code. Et nous a donné l'occasion de découvrir énormément de problèmes liés notamment à l'analyse, qui n'étaient pas aussi évidents dans la théorie du cours. Nous estimons avoir passé environ 3 semaines complète pour l'achèvement de ce compilateur. Les deux premiers jours on s'est focalisé sur le regroupement de la grammaire, ensuite les tâches se sont divisées en trois, chacun à son analyseur. Nous ne pensions pas passer autant de temps sur la conception de la grammaire, qui nous a valu de recommencer plusieurs fois et de passer de nombreuses heures à chercher la bonne méthode pour lever les ambiguïtés, sans en soulever d'autres. La difficulté en général réside dans l'implémentation de la grammaire lors la réalisation de l'analyseur syntaxique, ensuite la sémantique pourra être facilement vérifié en se basant sur la table de symbole généré par l'analyseur syntaxique.