École Nationale Supérieure d'Informatique

et d'Analyse des Systèmes

Rapport du projet :

Réalisation d'un compilateur de langage de programmation LOGO

Filière : Génie logiciel

Encadré par :
Pr. Youness TABII
Pr. Rachid OULAD HAJ THAMI

Réalisé par :
Ayoub HMADOUCH
Abderahmane KOTBI
Abdessamad EL HAFI
Jaouhar DERROUICH
Mohammed OUTALLEB

[Année universitaire 2021 – 2022]

Remerciements

Nous souhaitons adresser nos remerciements aux personnes qui nous ont apporté leur aide et qui ont contribué à la réalisation de notre projet.

Nous tenons à remercier du fond du cœur notre cher professeur monsieur **TABII Youness** de nous avoir assuré le cours de compilation qui nous a aider à bien concevoir et réaliser notre projet. Aussi, nous tenons à remercier également notre cher professeur monsieur **OULAD HAJ THAMI Rachid** qui nous a formé et accompagné tout au long des travaux pratiques de ce cours avec beaucoup de patience et de pédagogie.

Enfin, Nous tenons à remercier infiniment toute l'équipe pédagogique de l'École Nationale Supérieure d'Informatique et d'Analyse des Système.

Tables de matières

Chapitre 1 : Présentation du langage LOGO

1. Introduction

Le langage de programmation **Logo** a pour but d'améliorer chez les enfants leur manière de penser et d'aborder la résolution de problèmes. Matérialisé sous la forme d'un petit robot, la Tortue Logo, ce langage est utilisé dans la résolution de problèmes. Le langage de programmation Logo permet de déplacer cette « tortue géométrique », qui est un stylo représenté sous la forme d'une tortue que les enfants peuvent bouger ou dessiner sur l'écran. Notre Objectif dans ce projet c'est de créer un compilateur de ce langage.

2. Conception du langage

2.1. Commandes du langage

Primitives	Interprétation	Exemple
FORWARD X	La tortue avance de X point sur sa direction	FORWARD 100
RIGHT X	La tortue change la direction d'un angle de x degrés vers sa droite.	RIGHT 90
LEFT X	La tortue change la direction d'un angle de x degrés vers sa gauche.	LEFT 90

REPEAT X [
  INSTRUCTIONS
]
	La tortue répète le code INSTRUCTIONS X fois.	REPEAT 4 [
		  FORWARD 100
		  RIGHT 90
		]

IF (CONDITION) [
  INSTRUCTIONS
]
ELSE [
  INSTRUCTIONS
]
	La tortue exécute le code INSTRUCTIONS selon la condition vérifiée.	IF (i < 20) [
		  FORWARD 100
		  RIGHT 90
		]
		ELSE [
		  LEFT 90
		  FORWARD 100
		]

DEFINE FONCTION NAME [
  INSTRUCTIONS
]
	Création d'une fonction	DEFINE FONCTION SQUARE [
		  REPEAT 4 [
		  FORWARD 100
		  RIGHT 90
		  ]
		]

USE FONCTION NAME	Appeler la fonction	USE FONCTION SQUARE

2.2. Unités lexicales du langage

```
* Mots clés = { IF,
                ELSE,
                REPEAT,
                FORWARD,
                LEFT,
                RIGHT,
                COLOR,
                DEFINE,
                USE,
                FONCTION,
                TRANSPARENT,
                RED,
                GREEN,
                BLACK,
                BLUE}
* Symboles spéciaux = {[,]}
* Opérateurs = {+,-,*,/}
```

2.3. Grammaire du langage

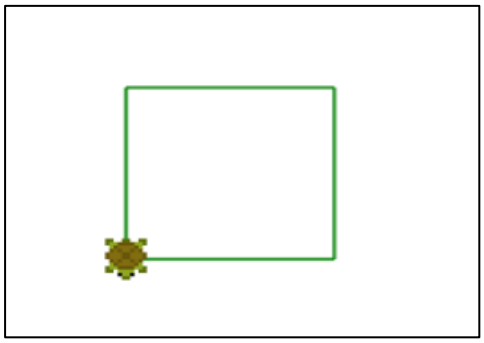
```
Soit  G = <T,NT,S,P> la grammaire de notre langage LOGO,les
règles de productions sont les suivantes:

<START> ::= <PROGRAM>

<PROGRAM> ::= <INSTRUCTION>
            | <PROGRAM> <INSTRUCTION>

<INSTRUCTION> ::= <FORWARD> <EXPRESSION>
                | <LEFT> <EXPRESSION>
                | <RIGHT> <EXPRESSION>
                | <REPEAT> <EXPRESSION> '[' <PROGRAM> ']'
                | <IF> <EXPRESSION> '[' <PROGRAM> ']' <BLOCK>
                | <DEFFONCTION> <EXPRESSION> '[' <PROGRAM> ']'
                | <RED>
                | <GREEN>
                | <BLUE>
                | <TRANSPARENT>
                | <BLACK>
                | <COLOR>
                | <USEFONCTION> <EXPRESSION>
```

```
<BLOCK> ::= <ELSE> '[' <PROGRAM> ']'
          | <empty>

<EXPRESSION> ::= <EXPRESSION> '+' <TERM>
               | <EXPRESSION> '-' <TERM>
               | <EXPRESSION> '*' <TERM>
               | <EXPRESSION> '/' <TERM>
               | TERM

<TERM> ::= <TERM> '*' <FACTOR>
         | <TERM> '/' <FACTOR>
         | <FACTOR>

<FACTOR> ::= <VALUE>
           | '-' <FACTOR>
           | '(' <EXPRESSION> ')'
```

2.4. Exemples du code logo

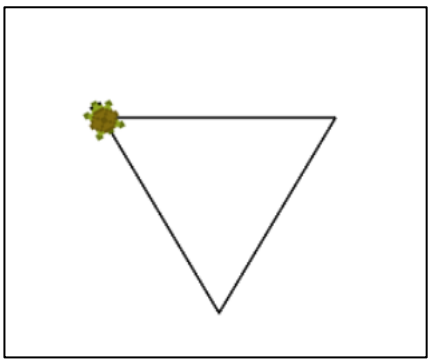
- **Tracer un carré :**

```
REPEAT 4 [
   FORWARD 100
   LEFT 90
]
```



- **Tracer un triangele :**

```
FORWARD 150
RIGHT 120
FORWARD 150
RIGHT 120
```

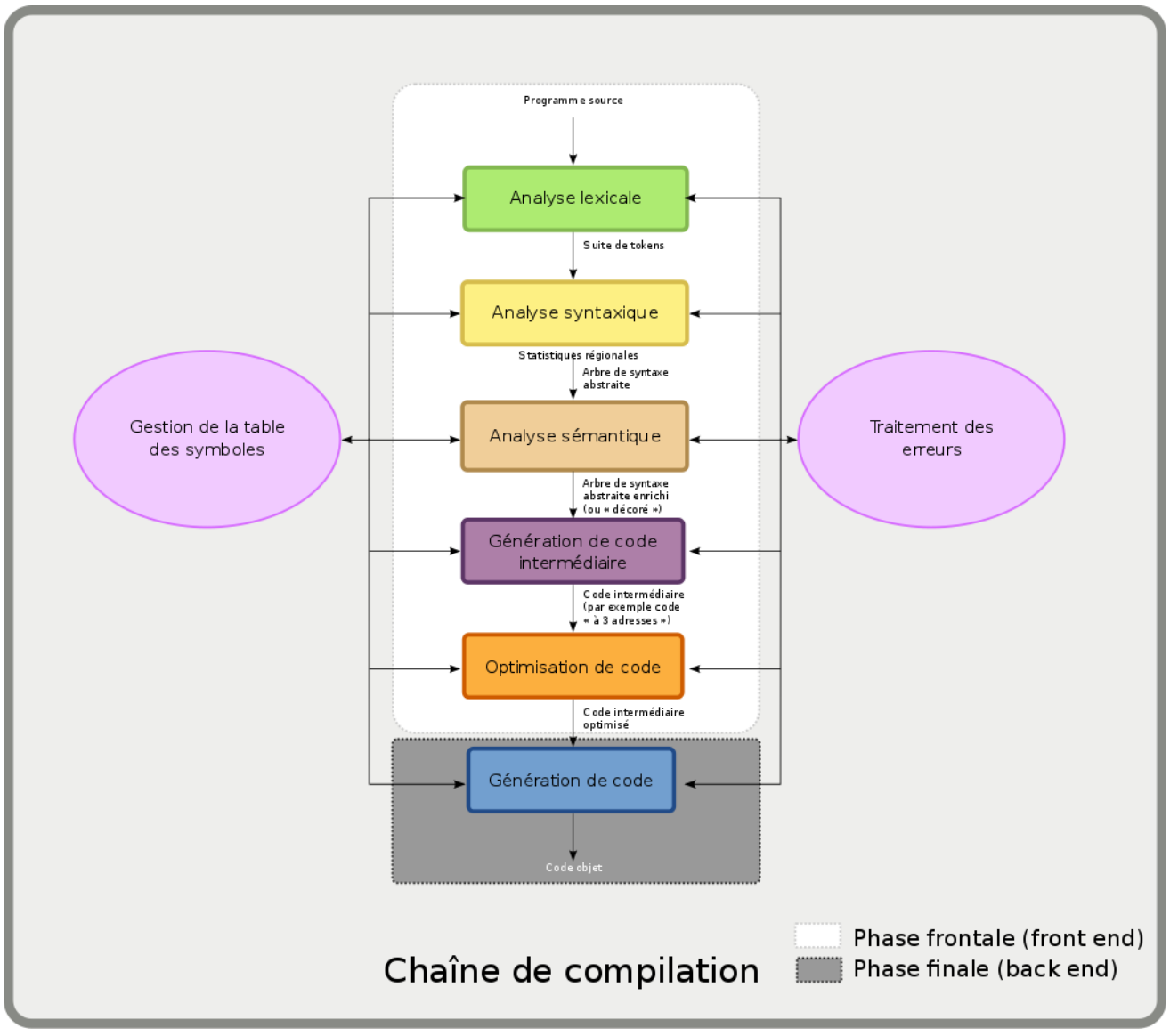


Chapitre 2 : Réalisation et test de compilateur

1. Chaîne de compilation

Un compilateur effectue les opérations suivantes : analyse lexicale, pré-traitement (préprocesseur), analyse syntaxique (parsing), analyse sémantique, et génération de code optimisé. La compilation est souvent suivie d'une étape d'édition des liens, pour générer un fichier exécutable. Quand le programme compilé (code objet) est exécuté sur un ordinateur dont le processeur ou le système d'exploitation est différent de celui du compilateur, on parle de compilation croisée.

La figure ci-dessous représente la chaine de compilation à suivre pour développer un compilateur d'un tel langage :



- **Analyse lexicale** : vérification que les caractères ou suites des caractères du programme sont des constituants d'un programme du langage cible (ici le langage C). Cette phase découpe le flux de caractères du fichier en une suite d'unités lexicales prise en compte par l'étape suivante.

- **Analyse syntaxique** : vérification que l'ordonnancement des unités lexicales respecte une configuration possible décrite par une grammaire.

- **Analyse sémantique** : vérification que les opérations décrites par la suite d'instructions ont un sens pour les données manipulées. Par exemple, comparer une donnée simple, comme un entier à un ensemble de données, comme les caractéristiques d'un individu, n'a pas de sens, même si la grammaire permet l'écriture d'une instruction semblable.

2. Outils de développement

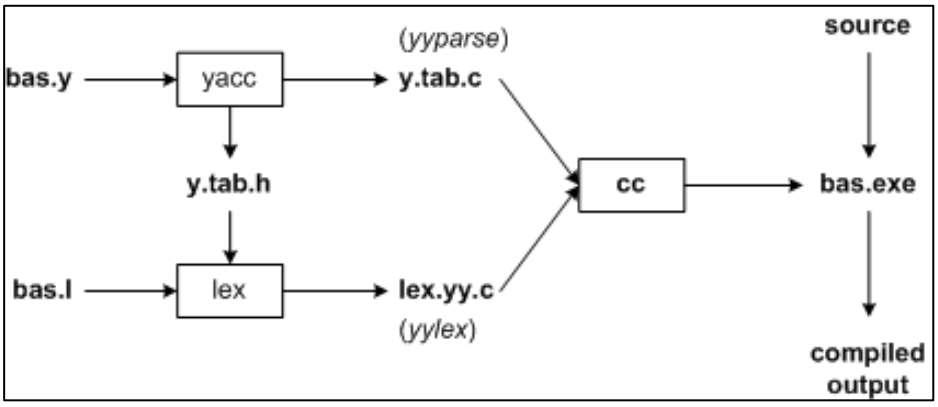
Pour réaliser ce compilateur nous avons utilisé pour ce but les deux générateurs d'analyseurs suivants :

2.1. Flex

Flex est un outil pour générer des analyseurs, programmes qui reconnaissent des motifs lexicaux dans du texte. Il lit les fichiers d'entrée donnés, ou bien son entrée standard si aucun fichier n'est donné, pour obtenir la description de l'analyseur à générer.

2.2. Yacc/Bison

Bison est un générateur d'analyseur syntaxique qui fait partie du projet GNU. Bison lit une spécification d'un langage sans contexte, avertit de toute ambiguïté d'analyse et génère un analyseur (en C, C ++ ou Java) qui lit les séquences de jetons et décide si la séquence est conforme à la syntaxe spécifiée par la grammaire.



3. Réalisation

3.1. Structure du projet

Dossier / Fichier	Rôle
Makefile	ce fichier contient les commandes de make permettant de générer les executables et les dessins
parser.y	code d'analyseur syntaxique et sémantique
scanner.l	code d'analyseur lexical
semantic.c	des fonctions sémantiques
semantic.h	les structures et les fonctions utilisés dans la partie sémantique
test	un ensemble de tests

3.2. Analyseur lexical

```
%}

%%

[1-9][0-9]*|0   { yylval.val=atoi(yytext); return VALUE;}
\[              {return '[';}
\]              {return ']';}
"REPEAT"        { return REPEAT;}
"IF"            { return IF;}
"FORWARD"       { return FORWARD;}
"RIGHT"         { return RIGHT;}
"LEFT"          { return LEFT;}
"BLUE"          { return BLUE;}
"GREEN"         { return GREEN;}
"RED"           { return RED;}
"TRANSPARENT"   { return TRANSPARENT;}
"BLACK"         { return BLACK;}
"SQUARE"        { return SQUARE;}
"COLOR"         {return COLOR;}
"STAR"          {return STAR;}
"0x"[1-9a-fA-F][0-9a-fA-F]* {yylval.val=strtol(yytext,NULL,0); return HEXA;}
"USE FONCTION"  {return USEFONCTION;}
"DEFINE FONCTION"   {return DEFFONCTION;}
\t              { /* ignorer */ }
\n              { /* ignorer */ }
\               { /* ignorer */ }

%%
```

3.3. Analyseur syntaxique

```
%token   FORWARD RIGHT LEFT REPEAT IF ELSE VALUE BLUE GREEN RED TRANSPARENT BLACK COLOR HEXA DEFFONCTION USEFONCTION


%union {
    NODE* NODE_TYPE;
    int val;
};

%type <NODE_TYPE> FORWARD RIGHT LEFT REPEAT IF ELSE BLOCK BLUE GREEN RED TRANSPARENT BLACK COLOR DEFFONCTION USEFONCTION
%type <NODE_TYPE> PROGRAM INSTRUCTION
%type  <val> EXPRESSION VALUE TERM FACTOR HEXA

%%

START : PROGRAM
{
    root = $1;
    YYACCEPT;
}

PROGRAM : INSTRUCTION
{
    $$ = $1;
}
| PROGRAM INSTRUCTION
{
    $$ = append_node($2, $1);
}

INSTRUCTION : FORWARD EXPRESSION
{
    $$ = create_node(FORWARD_TOKEN, $2, NULL);
}
| LEFT EXPRESSION
{
    $$ = create_node(LEFT_TOKEN, $2, NULL);
}
| RIGHT EXPRESSION
{
    $$ = create_node(RIGHT_TOKEN, $2, NULL);
}
| REPEAT EXPRESSION '[' PROGRAM ']'
{
    NODE *rep = create_node(REPEATc, $2, NULL);
    $$ = append_node_repeat($4, rep);
}
| IF EXPRESSION '[' PROGRAM ']' BLOCK
{
    NODE *ift = create_node(IF_TOKEN, $2, NULL);
    $$ = NULL;
    $$ = append_node_if($4, ift);
    $$ = append_node($6, $$);
}
| DEFFONCTION EXPRESSION '[' PROGRAM ']'
{
    NODE *def = create_node(DEFFONCTION_TOKEN, $2, NULL);
    $$ = append_node_repeat($4, def);
}
| RED
{
    $$ = create_node(RED_TOKEN, 0, NULL);
}
| GREEN
{
    $$ = create_node(GREEN_TOKEN, 0, NULL);
}
```

```
| BLUE
{
    $$ = create_node(BLUE_TOKEN, 0, NULL);
}
| TRANSPARENT
{
    $$ = create_node(TRANSPARENT_TOKEN, 0, NULL);
}
| BLACK
{
    $$ = create_node(BLACK_TOKEN, 0, NULL);
}
| COLOR HEXA
{
    $$ = create_node(COLOR_TOKEN, $2, NULL);
}
| USEFONCTION EXPRESSION
{
    $$ = create_node(USEFONCTION_TOKEN, $2, NULL);
}

BLOCK : ELSE '[' PROGRAM ']'
{
    NODE *elst = create_node(ELSE_TOKEN, 0, $3);
    $$ = NULL;
    $$ = append_node_if($$, elst);
}
```

```
| %empty
{
    $$ = NULL;
    printf("\n");
}

EXPRESSION : EXPRESSION '+' TERM
{
    $$ = $1 + $3;
}
| EXPRESSION '-' TERM
{
    $$ = $1 - $3;
}
| TERM
{
    $$ = $1;
}

TERM : TERM '*' FACTOR
{
    $$ = $1 * $3;
}
| TERM '/' FACTOR
{
    $$ = $1 / $3;
}
```

```
| FACTOR
{
    $$ = $1;
}

FACTOR : VALUE
{
    $$ = $1;
}
| '-' FACTOR
{
    $$ = -$2;
}
| '(' EXPRESSION ')'
{
    $$ = $2;
}

%%

int main(){

    yyparse();
    int a=0;
    print_node(root, &a);
    draw(root);
    free_node(&root);
    return 0;
}
```

3.4. Analyse sémantique

Afin de bien réaliser la partie sémantique, il est nécessaire de créer un ensemble de fonctions et de structures qui permettent d'analyser le programme. En ce qui concerne notre implémentation, l'analyse sémantique va lire et afficher le programme donnée. Pour le faire, on a implémenté :

- **Les structures suivantes :**

```
enum token
{
    FORWARD_TOKEN,
    LEFT_TOKEN,
    RIGHT_TOKEN,
    REPEATc,
    IF_TOKEN,
    ELSE_TOKEN,
    BLUE_TOKEN,
    GREEN_TOKEN,
    RED_TOKEN,
    TRANSPARENT_TOKEN,
    BLACK_TOKEN,
    COLOR_TOKEN,
    USEFONCTION_TOKEN,
    DEFFONCTION_TOKEN
};

typedef enum token token;
```

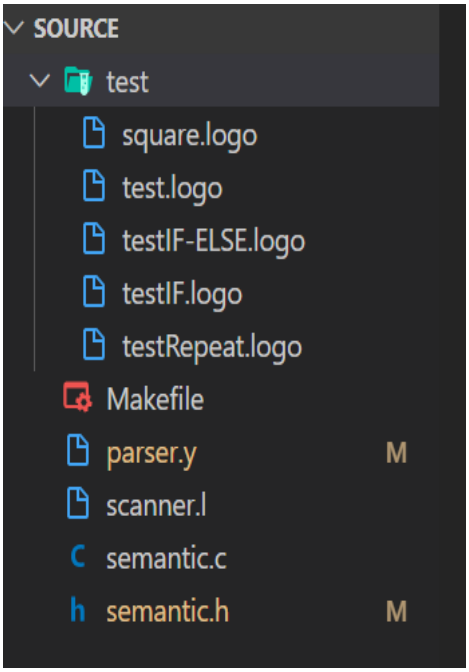
```
Kotbi Abderrahmane, 2 hours ago | 1 author (Kotbi Abderrahmane)
typedef struct node
{
    token intstruction;
    int value;
    struct node *program;
    struct node *next;
} NODE;

typedef NODE *INSTRUCTION;
Kotbi Abderrahmane, 2 hours ago | 1 author (Kotbi Abderrahmane)
typedef struct action
{
    double left;
    double right;
    double down;
    double up;
} ACTION;
```

- **<u>Les fonctions suivantes :</u>**

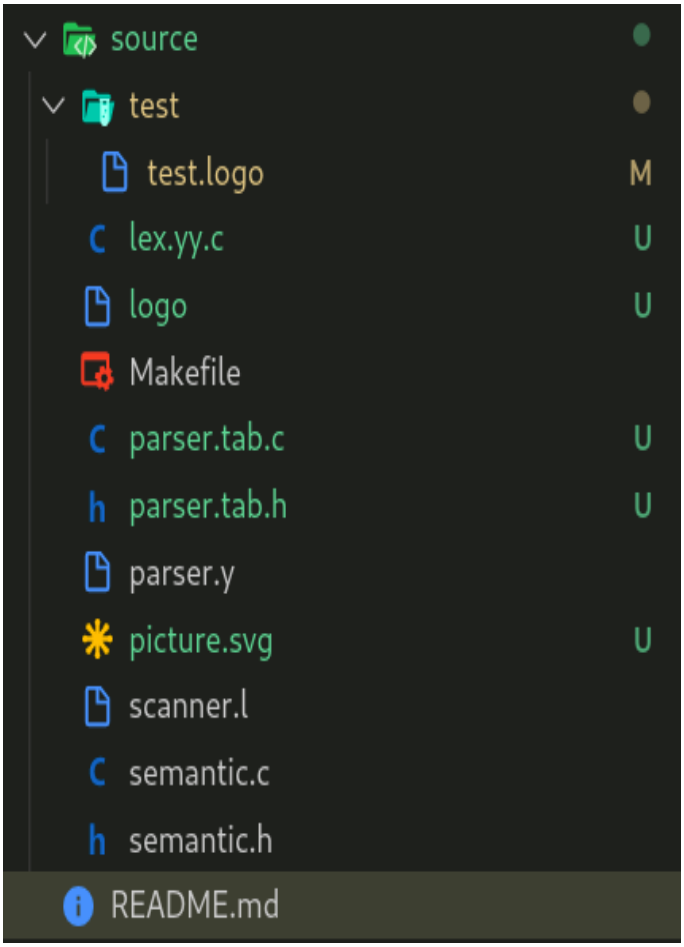
```
//create the node
NODE *create_node(token instruction, int value, NODE *program);
//append the node to the linkedlist
NODE *append_node(NODE *pn1, NODE *pn2);
//append the node repeat to the linkedlist
NODE *append_node_repeat(NODE *pn1, NODE *pn2);
//append the node if to the linkedlist
NODE *append_node_if(NODE *pn1, NODE *pn2);
//print the node
void print_node(NODE *pn, int *tab);
//free the memoire
void free_node(NODE **n);
void draw(NODE *n);
void draw_line(FILE *file, NODE *n, NODE *start);
void update_svg(NODE *n, NODE *start, ACTION *value);
void show_struct(ACTION movment);
double max(double a, double b);
double absolute_value(double a);
void write_line(FILE *file, double x2, double y2);
```

3.5. Organisation du projet

- Le projet avant de générer les exécutables a la structure suivante :



- Le projet après la génération des exécutables a la structure suivante :



3.6. Commandes de compilation

Pour faciliter la compilation de projet nous avons utilisé un makefile qui contient l'ensemble des commandes nécessaires :

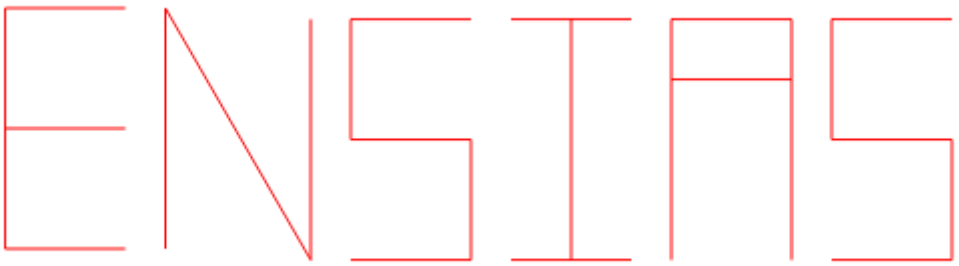
```
run: build

build: lex.yy.c parser.tab.c semantic.c semantic.h
    gcc lex.yy.c parser.tab.c semantic.c -o logo -lm

parser.tab.c: parser.y
    bison -d parser.y

lex.yy.c: scanner.l
    flex scanner.l

semantic: semantic.o
    gcc semantic.o -o semantic

semantic.o: semantic.c
    gcc -c semantic.c -o semantic.o

draw: build
    ./logo < test/test.logo

clean:
    @rm -f *.o parser.tab.c parser.tab.h lex.yy.c
    @rm -f logo
    @rm -f *.svg
```

Donc pour compiler le projet avec l'exemple de code logo test.logo, il faut juste utiliser la commade : make draw

4. Tests

Le dossier test dans le projet contient un ensemble de tests de code logo.

- Considérons le code du fichier testEnsias.logo, le compilateur va retourner le résultat suivant :
  - Le code compilé sans erreurs.
  - Le dessin en format SVG.



Conclusion

Ce projet nous a permis d'aller plus loin dans notre compréhension des différents types d’analyse : lexicale, syntaxique et sémantique. Il nous a permis également de voir un exemple concret et d'appliquer les principes théoriques vus en cours. Nous avons désormais une vision globale plus claire sur les différentes phrases allant de l'écriture à l'exécution d'un programme.