Table des matières

1 Introduction 3
1.1 Possibilités du translateur dot2d3........ 3
1.2 Choix des outils 3
2 La structure et l’utilisation du translateur dot2d3........ 4
2.1 Structure des fichiers sources 4
2.2 Compilation du projet........ 5
2.3 Lancement du translateur dot2d3........ 5
3 Réalisation du translateur dot2d3 6
3.1 Front-end 6
3.1.1 Analyse lexicale........ 6
3.1.2 Analyse syntaxique........ 7
3.2 Middle-end........ 11
3.3 Back-end 13
4 Test du translateur dot2d3 13
5 Exemples des résultats obtenus 14
6 Références........ 18

1 Introduction

Le but de ce projet est d’écrire un programme (un translateur) qui prend en entrée un fichier au format dot et qui produit du code JavaScript qui permet d’afficher ce graphe, en utilisant la bibliothèque D3.js.

1.1 Possibilités du translateur dot2d3

La grammaire du langage dot :

```
      graph : [ strict ] (graph | digraph) [ ID ] '{' stmt_list '}'
  stmt_list : [ stmt [ ';' ] [ stmt_list ] ]
       stmt : node_stmt
            | edge_stmt
            | attr_stmt
            | ID '=' ID
            | subgraph
  attr_stmt : (graph | node | edge) attr_list
  attr_list : '[' [ a_list ] ']' [ attr_list ]
     a_list : ID '=' ID [ (';' | ',') ] [ a_list ]
  edge_stmt : (node_id | subgraph) edgeRHS [ attr_list ]
    edgeRHS : edgeop (node_id | subgraph) [ edgeRHS ]
  node_stmt : node_id [ attr_list ]
    node_id : ID [ port ]
       port : ':' ID [ ':' compass_pt ]
            | ':' compass_pt
   subgraph : [ subgraph [ ID ] ] '{' stmt_list '}'
 compass_pt : (n | ne | e | se | s | sw | w | nw | c | _)
```

Pour ce projet, on prend en compte un sous-ensemble de cette grammaire dans lequel :

- on se contentera des graphes non orientés ;
- on ne considérera pas les chaînes HTML comme identifiant ;
- on ne permettra pas de retour à la ligne dans les chaînes entre " avec le symbole \ suivi d’un retour chariot ;
- on se contentera des caractères ASCII ; et
- on traitera les compass_pt comme n’importe quel identifiant.

1.2 Choix des outils

Pour la réalisation de ce projet, le langage C/C++ a été utilisé.
Les outils flex/bison, les versions GNUs de lex/yacc, ont été utilisé pour l’analyse lexicale et syntaxique.

Pour la partie Middle-end et Back-end on utilise les classes de la bibliothèque STL (notamment les classes string, vector, map, pair).
On utilise aussi le logiciel xDot pour visualiser le graphe au format dot et le comparer avec le résultat obtenu d'après le translateur réalisé.

2 La structure et l'utilisation du translateur dot2d3

2.1 Structure des fichiers sources

La structure des fichiers sources du projet est présentée dans le tableau ci-dessous :

Nom du fichier / dossier	Description
readme	fichier readme
tests/	dossier contenant des fichiers d'entrée .dot pour tester le projet (voir la rubrique "Test du translateur dot2d3")
src/	dossier contenant les fichiers sources du translateur
src/flex.f	fichier d'entrée pour flex
src/bison.y	fichier d'entrée pour bison
src/nodeStucts.h	déclarations des types des nœuds
src/createStructsFunctions.h, src/createStructsFunctions.c	fonctions de création des nœuds
src/jsgraph.h, src/jsgraph.cpp	classe utilisée pour le Middle-end et Back-end (voir les rubriques correspondantes)
src/main.cpp	fonction main
src/js_epilog, src/js_prolog	fichiers communs pour la génération du JavaScript
src/d3.min.js	bibliothèque D3.js
src/Makefile	Makefile pour la compilation du projet
src/runtests.sh	shell script pour tester
src/test.dot	fichier .dot d'entrée

2.2 Compilation du projet

Pour simplifier la compilation du projet, le Makefile a été créé :

```
all: dot2d3 clean

dot2d3: bison.tab.o lex.yy.o createStructsFunctions.o jsgraph.o main.o
	g++ -o dot2d3 bison.tab.o lex.yy.o createStructsFunctions.o jsgraph.o main.o
-lfl

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

lex.yy.c:
	flex flex.f

lex.yy.o: lex.yy.c
	gcc -c -w lex.yy.c

bison.tab.o: bison.tab.c
	gcc -c -w bison.tab.c

createStructsFunctions.o: createStructsFunctions.c
	gcc -c -w createStructsFunctions.c

jsgraph.o: jsgraph.cpp
	g++ -c -w jsgraph.cpp

main.o: main.cpp
	g++ -c -w main.cpp

clean:
	rm -rf *.o lex.yy.c bison.tab.h bison.tab.c
```

Pour compiler le projet, utilisez la commande *make all*. A la sortie vous obtiendrez un fichier exécutable dot2d3.

2.3 Lancement du translateur dot2d3

Commande pour lancer le translateur : *dot2d3 <nom_du_fichier_dot>*. A la sortie on obtient un fichier html qui contient un graphe au format D3 *<nom_du_fichier_dot>.html*.

3 Réalisation du translateur dot2d3

3.1 Front-end

L'objectif de cette phase est de reconnaître un fichier au format dot et de produire l'arbre de syntaxe abstraite correspondant. Cette étape contient deux sous-étapes : l'analyse lexicale et l'analyse syntaxique.

3.1.1 Analyse lexicale

Pour faire une analyse lexicale, le fichier flex.f a été créé :

```
/* definitions */
%{
        #include "bison.tab.h"
        extern int yyparse(void);
        #define YY_USER_ACTION yylloc.first_line = yylloc.last_line = yylineno;

        char* str[1000]; // Buffer for string constants and commentairs
        int errNum = 0;  // Errors number
%}

%option noyywrap
%option never-interactive
%option yylineno
%x STRING
%x COMMENT



%% /* rules */

(?i:graph)                {return GRAPH;}          /* ?i means case-independent */
(?i:subgraph)            {return SUBGRAPH;}
(?i:strict)               {return STRICT;}
(?i:node)                 {return NODE;}
(?i:edge)                 {return EDGE;}

\=               {return '=';}
\[               {return '[';}
]                {return ']';}
\{               {return '{';}
\}               {return '}';}
--               {return LINK;}
;                {return ';';}
,                {return ',';}


[-]?[0-9]*\.?[0-9]+      {   // number (we deal with it like with id)
                             yylval.Id = (char*)malloc(strlen(yytext)+1);
                             strcpy(yylval.Id, yytext);
                             return ID;
                         }

[_a-zA-Z][_a-zA-Z0-9]*   {   // string id
                             yylval.Id = (char*)malloc(strlen(yytext)+1);
```

```
                            strcpy(yylval.Id, yytext);
                            return ID;
                        }

[ \t\n]*#.*                    { /* Found # comment, return nothing */ }
[ \t\n]*\/\/.*                 { /* Found // comment, return nothing */ }

"/*" BEGIN(COMMENT); str[0]=0;
<COMMENT>[^*]* strcat(str,yytext);
<COMMENT>"*"+[^*/]* strcat(str,yytext);
<COMMENT>"*"+"/" {          /* Found multi-line comment, return nothing */
                                    BEGIN(INITIAL);
                        }

\" BEGIN(STRING); str[0]=0;
<STRING>[^\"\\]+ strcat(str,yytext);
<STRING>\\ strcat(str,"\\");
<STRING>\\\" strcat(str,"\"");
<STRING>\"                  {
                                    BEGIN(INITIAL);
                                    // Found string constant
                                    yylval.Id = (char*)malloc(strlen(str)+1);
                                    strcpy(yylval.Id, str);
                                    return ID;
                            }

[\t \n\r]
.        { printf("Line %d: bad character \"%c\"\n", yylineno, yytext); ++errNum; }


%% /* user subroutines */
/* empty */
```

Pour trouver les commentaires entre /* et */ et les chaînes de caractères les start conditions ont été utilisées (*COMMENT* et *STRING* respectivement).

3.1.2 Analyse syntaxique

Selon la grammaire du langage dot, on a défini des structures pour chaque type de nœuds (le fichier nodeStructs.h) :

```
// Node graph
struct NGraph {
        int                        isStrict;        // true if the graph is stict
        char                       *id;             // graph id
        struct NStmtList           *stmtList;       // graph stmt list
};

// Node subgraph
struct NSubgraph {
        char                       *id;     // subrgraph id
        struct NStmtList           *stmts;  // subrgraph stmt list
};

// Node statement list
struct NStmtList {
```

```
        struct NStmt              *first;  // first statement in statement list
        struct NStmt              *last;   // last statement in statement list
};

// Node id (ex.: A)
struct NNodeId {
        char                      *id;     // node id
};

// Pair of ids (help structure for AList)
struct NIdPair {
        char                      *id1;    // first id
        char                      *id2;    // second id
        struct NIdPair            *next;   // next idPair in NAList (if exist)
};

// Node AList
// Ex.: color=blue; label="mynode", size=big
struct NAList {
        struct NIdPair            *first;  // first id pair in NAttrList
        struct NIdPair            *last;   // last id pair in NAttrList
        struct NAList             *next;   // next NAList in NAttrList (if exist)
};

// Node list of attributes
// Ex.: [a_list] [a_list] [a_list]
struct NAttrList {
        struct NAList             *first;          // first AList
        struct NAList             *last;           // last AList
};

// Statement types in NEdgeRhs node
enum EdgeRhsType {
        NodeIdRhs,        // node : A
        SubgraphRhs       // subgraph : {A--B--C}
};
// Node Edge RHS
// Ex.: --A or --{C D E}
struct NEdgeRhs {
        enum EdgeRhsType type;       // type of EdgeRhs
        struct NNodeId   *nodeId;    // node id (if node represents NodeIdRhs)
        struct NSubgraph *subgraph;  // subgraph (if node represents SubgraphRhs)
        struct NEdgeRhs  *next;      // next NEdgeRhs (if exist)
};

// Statement types in NStmt node
enum StmtType {
        Edge,             // edge statement type
        Node,             // node statement type
        Id,               // id statement type
        Subgraph,         // subgraph statement type
        Attr              // attr statement type
};

// General statement
struct NStmt {
        enum StmtType type;               // statement type

        struct NEdgeStmt     *edgeStmt;    // edge statement (for Edge type)
        struct NNodeStmt     *nodeStmt;   // node statement (for Node type)
        struct NIdStmt       *idStmt;     // id statement (for Id type)
        struct NSubgraph     *subraphStmt; // subgraph statement (for Subgraph type)
```

```
        struct NAttrStmt     *attrStmt;   // attr statement (for Attr type)

        struct NStmt         *next;       // next statement in a NStmtList
};

// Edge statement (one of - nodeId or subgraph - is filled according to node type)
// Ex.: A--<edgeRHS>[<attr_list>] or {A B C}--<edgeRHS>[<attr_list>]
struct NEdgeStmt {
        struct NNodeId           *nodeId;          // node id
        struct NSubgraph         *subgraph;        // subgraph
        struct NAttrList         *attrList;        // attributes (optional)
        struct NEdgeRhs          *edgeRhs;         // edge RHS (obligatoiry)
};

// Node statement
struct NNodeStmt {
        struct NNodeId           *nodeId;          // node id
        struct NAttrList *attrList;       // attributes (optional)
};

// ID=ID statement
struct NIdStmt {
        char                     *id1;             // first id in pair
        char                     *id2;             // second id in pair
};

// Attrubute types in NAttrStmt
enum AttrType {
        GraphAttr,      // graph (subgraph) attributes
        NodeAttr,       // node attributes
        EdgeAttr        // edge attributes
};

// Node attr statement
struct NAttrStmt {
        enum    AttrType type;             // type of attr statement
        struct  NAttrList        *attrList;        // attributes
};
```

Pour la création des nœuds, on a défini une liste des fonctions correspondantes (voir les fichiers createStructsFunctions.h, createStructsFunctions.c).
Après, le fichier d'entrée pour le bison a été créé (bison.y). Ici pour les règles de grammaire on utilise les fonctions de création de nœuds :

```
%{
        /*Prolog*/
        #include <stdio.h>
        #include <malloc.h>
        #include "nodeStructs.h"
        #include "createStructsFunctions.h"

        #define YYERROR_VERBOSE 0

        extern int yylex(void);
        struct NGraph *root;      // root of syntax tree
%}

%union {
        char *Id;
```

```
        struct NGraph *Graph;
        struct NSubgraph *Subgraph;
        struct NStmtList *StmtList;
        struct NNodeId *NodeId;
        struct NAList *AList;
        struct NAttrList *AttrList;
        struct NEdgeRhs *EdgeRhs;
        struct NStmt *Stmt;
        struct NEdgeStmt *EdgeStmt;
        struct NNodeStmt *NodeStmt;
        struct NAttrStmt *AttrStmt;
}

%locations

%type <Graph> graph
%type <StmtList> stmt_list
%type <Stmt> stmt
%type <NodeStmt> node_stmt
%type <EdgeStmt> edge_stmt
%type <AttrStmt> attr_stmt
%type <Subgraph> subgraph
%type <AttrList> attr_list
%type <AList> a_list
%type <NodeId> node_id
%type <EdgeRhs> edgeRHS

%token <Id> ID
%token GRAPH
%token SUBGRAPH
%token STRICT
%token LINK
%token NODE
%token EDGE


%%
graph: STRICT GRAPH ID '{' stmt_list '}'     {$$ = root = CreateGraph(1, $3, $5);}
| GRAPH ID '{' stmt_list '}'                 {$$ = root = CreateGraph(0, $2, $4);}
| STRICT GRAPH '{' stmt_list '}'             {$$ = root = CreateGraph(1, "", $4);}
| GRAPH '{' stmt_list '}'                    {$$ = root = CreateGraph(0, "", $3);}
;

stmt_list: /*empty*/                         {$$ = CreateStmtList();}
| stmt ';' stmt_list                         {$$ = AddToStmtList($1, $3);}
| stmt stmt_list                             {$$ = AddToStmtList($1, $2);}
;

stmt: node_stmt                              {$$ = CreateStmtN($1);}
| edge_stmt                                  {$$ = CreateStmtE($1);}
| attr_stmt                                  {$$ = CreateStmtA($1);}
| ID '=' ID                                  {$$ = CreateStmtI($1, $3);}
| subgraph                                   {$$ = CreateStmtS($1);}
;

attr_stmt: GRAPH attr_list                   {$$ = CreateAttrStmt(GraphAttr, $2);}
| NODE attr_list                             {$$ = CreateAttrStmt(NodeAttr, $2);}
| EDGE attr_list                             {$$ = CreateAttrStmt(EdgeAttr, $2);}
;

a_list: /*empty*/                            {$$ = CreateAList();}
```

```
| ID '=' ID ';' a_list                    {$$ = AddToAList($1, $3, $5);}
| ID '=' ID ',' a_list                    {$$ = AddToAList($1, $3, $5);}
| ID '=' ID a_list                        {$$ = AddToAList($1, $3, $4);}
;

edge_stmt: node_id edgeRHS attr_list      {$$ = CreateEdgeStmt($1, NULL, $2, $3);}
| node_id edgeRHS                         {$$ = CreateEdgeStmt($1, NULL, $2,
NULL);}
| subgraph edgeRHS attr_list              {$$ = CreateEdgeStmt(NULL, $1, $2, $3);}
| subgraph edgeRHS                        {$$ = CreateEdgeStmt(NULL, $1, $2,
NULL);}


edgeRHS: LINK node_id edgeRHS             {$$ = AddToEdgeRhsNodeId($2, $3);}
| LINK node_id                            {$$ = CreateEdgeRhsNodeId($2);}
| LINK subgraph edgeRHS                   {$$ = AddToEdgeRhsSubgraph($2, $3);}
| LINK subgraph                           {$$ = CreateEdgeRhsSubgraph($2);}
;

attr_list: '[' a_list ']' attr_list       {$$ = AddToAttrList($2, $4);}
| '[' a_list ']'                          {$$ = CreateAttrList($2);}
;

node_stmt: node_id attr_list              {$$ = CreateNodeStmt($1, $2);}
| node_id                                 {$$ = CreateNodeStmt($1, NULL);}
;

node_id: ID                               {$$ = CreateNodeId($1);}
;

subgraph: SUBGRAPH ID '{' stmt_list '}'   {$$ = CreateSubgraph($2, $4);}
| SUBGRAPH '{' stmt_list '}'              {$$ = CreateSubgraph("", $3);}
| '{' stmt_list '}'                       {$$ = CreateSubgraph("", $2);}
;


%%
int yyerror(char* s) {
        printf("Line %d: %s\n", yylloc.first_line, s);
}
```

3.2 Middle-end

Le but de cette partie est de partir de l'arbre de syntaxe abstraite et de produire des ensembles de nœuds et des ensembles d'arêtes pour faciliter la génération du code cible javascript.
Pour la réalisation de cette partie (et de Back-end aussi), la classe JsGraph a été créé (voir jsgraph.h, jsgraph.cpp). Sa partie publique contient :

- un constructeur *JsGraph(NGraph *root)* qui prend en entrée un arbre de syntaxe abstraite et qui produit des ensembles de nœuds et des ensembles d'arêtes ;
- une méthode *void print_graph(std::string filename)* pour générer le javascript (voir la rubrique Back-end).

Les données de la classe (partie privée) sont présentées ci-dessous :

Nom	Description
char *m_id;	l'identifiant du graphe
bool m_isStrict	graphe stricktness Ex. : **strict** graph a {...}
std::vector<struct JsEdge> m_edges	une liste des arêtes du graphe
std::map < std::string, std::map <std::string, std::string> > m_nodes;	une liste des nœuds du graphe
std::map <std::string, std::string> m_graphAttr	une liste des liste des attributs 'globales' du graphe Ex. : graph[bgcolor=gray]
std::map <std::string, std::string> m_graphNodeAttr	les attributs 'globales' des nœuds du graphe Ex. : node[color=red]
std::map <std::string, std::string> m_graphEdgeAttr	une liste des attributs 'globales' des arêtes du graphe Ex. : node[label="edge_label"]

La structure représentant une arête du graphe (utilisée pour stocker les arêtes dans *m_edges*) :

```
struct JsEdge{
        std::string                             node1;   // sourse node id
        std::string                             node2;   // destination node id
        std::map <std::string, std::string>     attr;    // edge attributes :
map(attr_name, attr_value)
};
```

Dans sa partie privée, la classe *JsGraph* contient aussi des méthodes auxiliaires facilitant l'analyse de l'arbre de syntaxe abstraite.
L'algorithme du Middle-end :

- parcourir l'arbre de syntaxe abstraite, produire des ensembles de nœuds et des ensembles d'arêtes avec ses attributs ;
- pour toutes les nœuds trouvés, appliquer les attributs 'globales' des nœuds du graphe (*m_graphNodeAttr*) ;
- pour toutes les arêtes trouvés, appliquer les attributs 'globales' des arêtes du graphe (*m_graphEdgeAttr*) ;
- pour toutes les nœuds trouvés, ajouter des attributs par défaut : *label="node_name", color="black"* (seulement si ces attributs n'étaient pas déclarés dans un fichier d'entrée *dot*) ;
- pour toutes les arêtes trouvés, ajouter un attribut par défaut : *color="black"* (seulement si cet attribut n'était pas déclaré dans un fichier d'entrée *dot*).

3.3 Back-end

Le but de cette partie est de produire une page HTML contenant du code JavaScript permettant de générer un élément SVG à l'aide de la bibliothèque D3.js.
Pour cette partie, les méthodes supplémentaires de la classe *JsGraph* ont été créées :

- *print_node* et *print_edge* qui prennent respectivement un nœud et une arête et qui affiche sur la sortie standard le code JavaScript correspondant à leur déclaration et leur ajout ;
- *print_graph* qui prend un ensemble de nœuds et un ensemble d'arêtes et qui affiche sur la sortie standard le code HTML+javascript qui permet de visualiser le graphe ;
- encode qui traduit une chaîne quelconque en identifiant javascript.

On prend les attributs des nœuds et des arêtes tels quels pour produire le code JavaScript. L'attribut *color* est adapté pour la bibliothèque D3.js (c'est à dire que les couleurs des arêtes et des nœuds de le graphe d3 vont être les mêmes que ceux de le graphe dot). Pour cela, la partie commune a été modifiée :

```
var link = svg.selectAll(".link").data(force.links())
    .enter().append("line")
    .attr("class", "link")
    .style("stroke-width", "2")
    .style("stroke", function(d) { return d.color; });
var node = svg.selectAll(".node").data(force.nodes())
    .enter().append("circle")
    .attr("class", "node")
    .attr("r", function(d) {return (10 + d.label.length * 5); })
    .style("stroke", function(d) { return d.color; })
    .style("stroke-width", "2")
    .attr("fill", "white")
    .call(force.drag);
```

Les restrictions de cette partie :

- le nombre maximal des arêtes entre les nœuds *A* et *B* est égal à 1, car la bibliothèque *dot3.js* ne les visualise pas par défaut (donc, le graphe est toujours '*strict'* en termes du langage dot) ;
- la bibliothèque *dot3.js* ne visualise pas les boucles par défaut, donc on ne les prend pas en compte (exemple du boucle pour le nœud a : *a--a*) ;
- on ne vérifie pas si les valeurs de l'attribut *color* sont correctes (on prend ces valeurs telles quelles).

4 Test du translateur dot2d3

Afin de tester le projet, le jeu de tests (les fichiers .dot) ainsi que le fichier shell runtests.sh ont été créé.
Pour exécuter runtests.sh, il faut d'abord compiler tous les codes à l'aide de la commande *make.* Le fichier runtests.sh est présenté ci-dessous :

```
#!/bin/bash

EXEPATH="./dot2d3"
TESTDIR="../tests/"
FILES=(a.dot b.dot c.dot genetic_programming.dot test1.dot test2.dot test3.dot
test4.dot)

# create folder
mkdir -p test_results
# copy d3.min.js to folder
cp ./d3.min.js ./test_results

# execute tests
for item in ${FILES[*]}
do
	${EXEPATH} "${TESTDIR}${item}"
	mv "${TESTDIR}${item}.html" ./test_results
done

echo "done!"
```

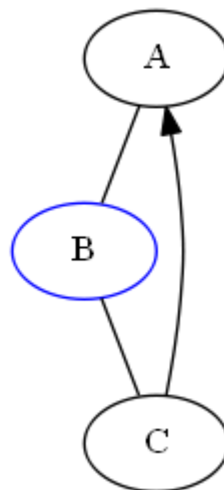
Ainsi, pour lancer tous les tests, il faut exécuter runtests.sh depuis un dossier src/. Les fichiers de sortie html vont être stockés dans un dossier src/testResults/.

5 Exemples des résultats obtenus

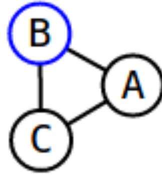
Test 1
Le fichier d'entrée : a.dot

```
graph a {
      A -- B -- C;
      C -- A [ dir = forward ];
      B [ color = blue ]
}
```

Le graphe dot correspondant (d'après xDot) :



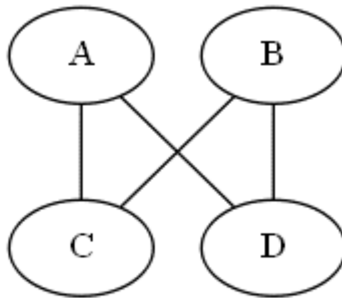
Le graphe d3 correspondant (d'après le translateur réalisé) :



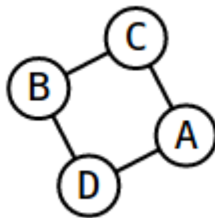
Test 2
Le fichier d'entrée : b.dot

```
graph b {
      { A B } -- subgraph { C D }
}
```

Le graphe dot correspondant (d'après xDot) :



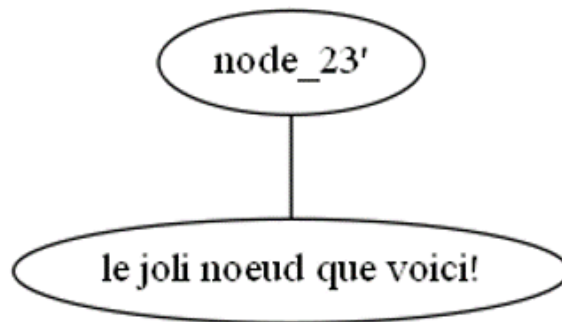
Le graphe d3 correspondant (d'après le translateur réalisé) :



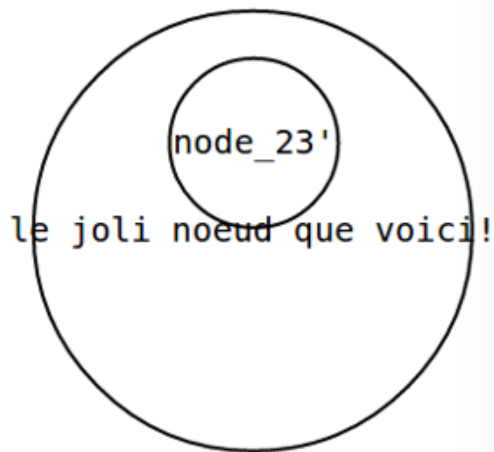
Test 3
Le fichier d'entrée : c.dot

```
graph c {
      "node_23'" -- "le joli noeud que voici!";
}
```

Le graphe dot correspondant (d'après xDot) :

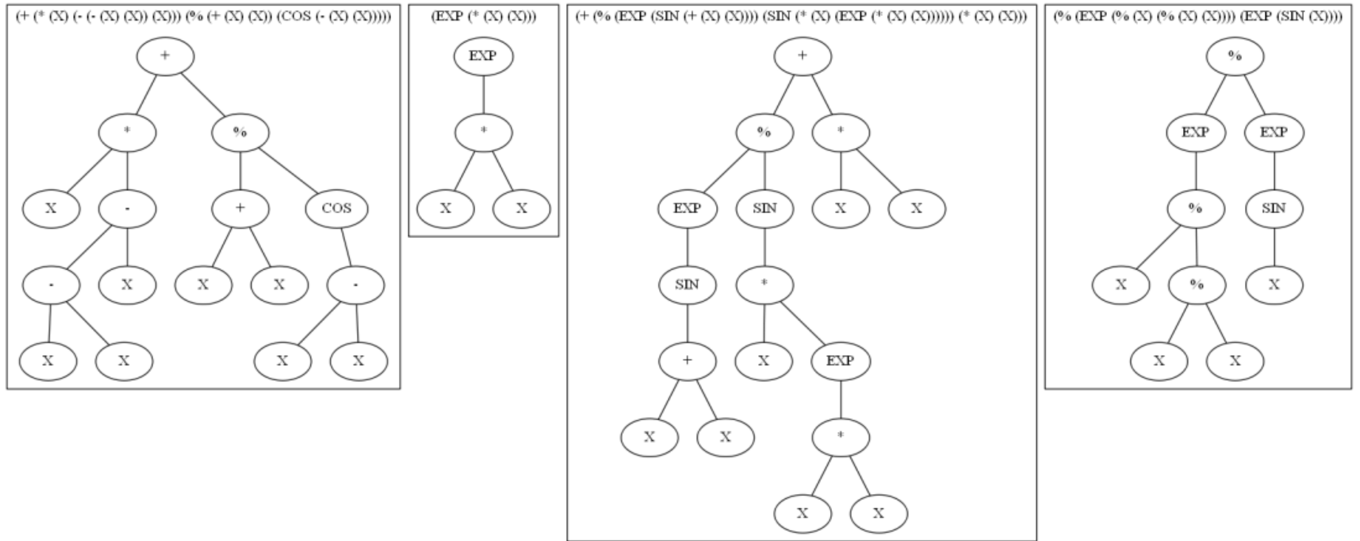


Le graphe d3 correspondant (d'après le translateur réalisé) :



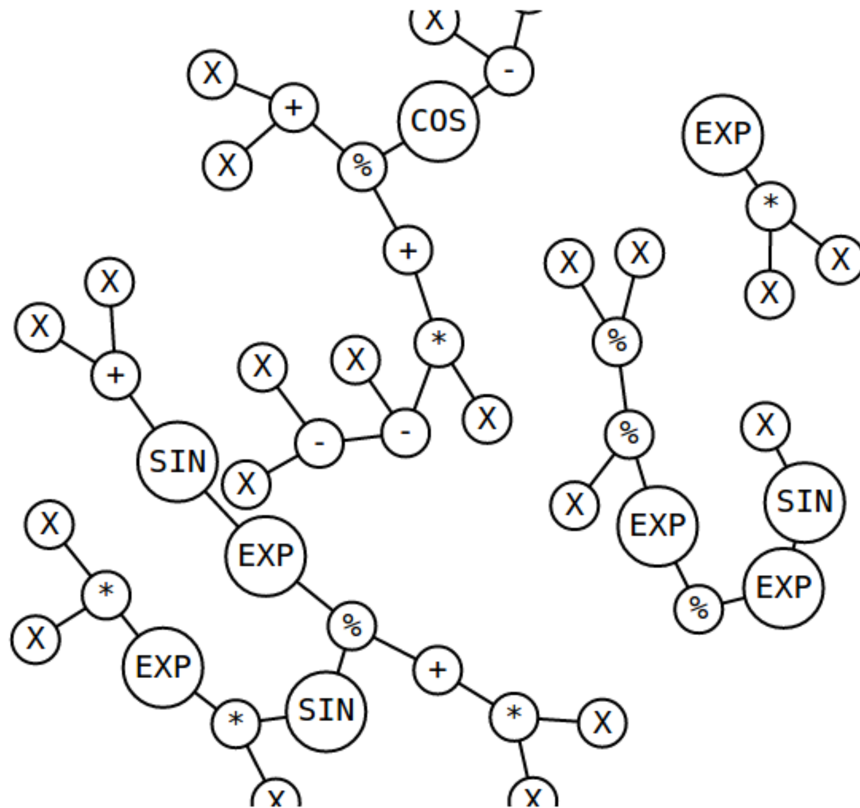
Test 4
Le fichier d'entrée : genetic_programming.dot (voir dans le dossier tests/).

Le graphe dot correspondant (d'après xDot) :



((+ (* (X) (- (- (X) (X)) (X))) (% (+ (X) (X)) (COS (- (X) (X))))) (EXP (* (X) (X))) (+ (% (EXP (SIN (+ (X) (X)))) (SIN (* (X) (EXP (* (X) (X)))))) (* (X) (X))) (% (EXP (% (X) (% (X) (X)))) (EXP (SIN (X)))))

Le graphe d3 correspondant (d'après le translateur réalisé) :



6 Références

1. La grammaire DOT : http://www.graphviz.org/doc/info/lang.html
2. La bibliothèque D3.js (site officiel) : http://d3js.org
3. Bison (documentation) : http://www.gnu.org/software/bison/
4. Flex (documentation) : http://flex.sourceforge.net
5. STL (documentation) : http://www.cplusplus.com/reference/