Générateur de Monde

JUILLIARD Quentin S6O7
COGNE Romain S6O6

Monsieur RABAT, INFO0602 / Département Informatique

JUILLIARD-COGNE

Jeudi 23 mars 2023

Table des matières

1 Introduction

Dans le cadre du module de Langages et Compilation (INFO0602) de 3ème année de licence informatique, il nous à été demandé de réaliser un projet qui a pour but de génerer des mondes pour un jeu de plateformes multi-joueurs en réseau.

2 Organisation générale

Dans cette section, nous verrons comment est organisé le projet, ainsi que comment le compiler et le démarrer.

2.1 Organisation des fichiers

Dans notre archive, une fois décompressée, vous trouverez de nombreux fichiers. Tout d'abord, nous avons fait le choix de ne pas faire de sous-dossiers pour simplifier le travail.

Néanmoins, il y a divers fichiers :

— .c : Des fichiers sources
— .h : des fichiers d'en-têtes
— .yacc : Fichier de grammaire Yacc
— .lex : Fichier d'analyse lexicale

Dans l'archive rendue, vous trouverez quatre dossiers. Le premier dossier est le Code C qui nous a été fournit. Le second dossier est intitulé *Code_Lex_Yacc_En_developpement* celui-ci regroupe l'ensemble des codes que nous avons réalisé avec une version finie qui comporte des bugs. Nous pouvons dire qu'il s'agit d'une version beta de nos codes. Le troisième dossiers est intitulé *Code_Lex_Yacc_fonctionnel* regroupe l'ensemble de nos codes testés et fonctionnels sur l'ensemble des fichiers test présents dans le quatrième dossier. Le quatrième dossier regroupe l'ensemble des fichiers de test utilisés dans notre projet.

2.2 Compilation et lancement

Pour lancer notre projet vous devez d'abord le compiler par la commande *make*. Puis par la suite une fois la compilation réalisée, utiliser la commande suivante : *./ lecteur_level <filename>*. Le filename doit absolument être un fichier en .txt pour avoir un résultat correct.

3 Structures utilisées

3.1 Structure d'un level

```
1  // Grid size
2  #define HEIGHT     20
3  #define WIDTH      60
4
5  // Structure of a level
6  typedef struct {
7      wint_t cells[HEIGHT][WIDTH];
8      color_t colors[HEIGHT][WIDTH];
9      block_t blocks[HEIGHT][WIDTH];
10 } level_t;
```

Listing 1 – Structure d'un level

La structure présente ci-dessus se nomme *level_t*, elle est utilisée dans notre projet pour gérer des niveaux dans un jeu.

Cette structure est composée de trois tableaux bidimensionnels nommés "cells", "colors" et "blocks" de dimensions "HEIGHT" par "WIDTH".

Le tableau *cells* est composé d'éléments de type *wint_t* qui représente des cellule présente sur la grille du niveau. Le tableau *colors* contient des éléments de type *color_t* qui stocke la couleur de chaque cellule. Le tableau *blocks* est composé d'élement de type *block_t* qui stockent les blocks présents dans chaque *cells*.

Nous pouvons observer la présence de deux variables globales intitulées *HEIGHT* et *WIDTH* qui représentent respectivement la hauteur et la largeur de la grille.

En conclusion, cette structure nous permet de stocker les informations nécessaires pour réprésenter un niveau de jeu pour chaque cellule de notre niveau.

3.2 Structure d'un bloc

```
typedef enum block_name_type{
    EMPTY,
    BLOCK,
    TRAP,
    LIFE,
    BOMB,
    DOOR,
    ENTER,
    EXIT,
    LADDER,
    ROBOT,
    PROBE,
    KEY,
    GATE
} block_name_t;

typedef struct block_type{
    block_name_t type;
    int value;
    int coordX;
    int coordY;
} block_t;
```

Listing 2 – Structure d'uun Block

Les structures présentes ci-dessus sont : une énumération nommée *block_name_type* et une structure nommée *block_t*.

L'énumération *block_name_type* définit différents types de blocs que nous pouvons retrouver dans un niveau du jeu.

La structure *block_t* contient quatre membrse. Le premier est le type du block présent sur la map de type *block_name_type*. Le second est la value par défaut, la value est à 0 néanmoins quand il s'agit d'une porte ou d'une clé sa valeur change, la value est de type *int*. Un bloc possède une position sur notre map en X et Y pour cela nous avons deux variables intitulées *coordX* et *coordY* qui sont de types *int*.

Pour conclure, la structure *block_t* permet de stocker les informations de chaque blocs présents sur la map du jeu.

3.3 Structure d'un symbole

```
typedef struct symbol_type {
    char* name;
    int value;
} symbol_t;
```

Listing 3 – Structure d'un symbole

Dans le code ci-dessus nommé *symbol_t*, nous avons deux membres. Le premier membre est un pointeur vers un tableau de caractères qui défini le nom du symbole. Le second membre est un entier qui correspond à la valeur d'affectation du symbole, il est de type int.

Néanmoins, il faut noter que le nom d'un symbole n'a pas de taille fixe vue qu'il s'agit d'un pointeur vers un char.

3.4 Structure d'une liste de symbole

```
typedef struct lst_symbol_type {
    symbol_t* symbol;
    struct lst_symbol_t* next;
} lst_symbol_t;
```

Listing 4 – Structure d'une liste de symbole

Le code décrit ci-dessus est une structure intitulé *lst_symbole_type*. Il se compose de deux membres. Le premier est *symbole*, il s'agit d'un pointeur vers une variable de type *symbol_t* et le second est *next*, il s'agit d'un pointeur qui pointe vers la prochaine structure de type *lst_symbole_type* dans la liste chaînée.

Nous réalisons donc une liste chaînée, une structure de données, qui lit les éléments entre eux par des pointeurs. Dans notre cas, chaque élément de cette liste est défini par une structure *lst_symbole_type* qui contient la référence du suivant grâce à la variable *next*.

Dans cette structure, nous stockons des éléments de type *symbole*.

3.5 Structure d'une table de symbole

```
typedef struct table_type {
    lst_symbol_t* list;
    lst_symbol_t* head;
} table_t;
```

Listing 5 – Structure d'une table de symbole

La structure *table_t* est une structure de données qui contient deux membres. Le premier membre est un pointeur de type *lst_symbol_t* qui pointe vers une liste chaînée de symboles. Le second champ est de même type que le premier et il pointe vers la tête de la liste.

La liste chaînée de symbole est stockée dans l'élément *list*. Elle est parcourue par l'utilisation du pointeur suivant. Le pointeur *head* pointe vers le 1er élément de la liste alors que le dernier élement de la liste à pour pointeur *next* la valeur NULL.

3.6 Structure d'une instruction

```
typedef enum instruction_type_type{
    CALL_PROCEDURE_INSTRUCTION,
    ASSIGNMENT_INSTRUCTION,
    CONDITIONAL_INSTRUCTION,
    WHILE_LOOP_INSTRUCTION,
    FOR_LOOP_INSTRUCTION,
    LADDER_INSTRUCTION,
    RECT_INSTRUCTION,
    FRECT_INSTRUCTION,
    HLINE_INSTRUCTION,
    VLINE_INSTRUCTION,
    GATE_INSTRUCTION,
    PUT_INSTRUCTION
} instruction_type_t;

typedef struct ladder_data_type{
    symbol_t* x;
    symbol_t* y;
    symbol_t* h;
} ladder_data_t;

typedef struct rect_data_type{
    symbol_t* x1;
    symbol_t* y1;
    symbol_t* x2;
    symbol_t* y2;
    char* block;
} rect_data_t;

typedef struct frect_data_type{
    symbol_t* x1;
    symbol_t* y1;
    symbol_t* x2;
    symbol_t* y2;
    char* block;
} frect_data_t;

typedef struct hline_data_type{
    symbol_t* x;
    symbol_t* y;
    symbol_t* l;
    char* block;
} hline_data_t;

typedef struct vline_data_type{
    symbol_t* x;
    symbol_t* y;
    symbol_t* l;
    char* block;
```

```
50 } vline_data_t;
51
52 typedef struct gate_data_type{
53     symbol_t*  x;
54     symbol_t* y;
55     symbol_t* n;
56 } gate_data_t;
57
58 typedef struct put_data_type{
59     symbol_t* x;
60     symbol_t* y;
61     char* block;
62 } put_data_t;
63
64 typedef struct instruction_type{
65     instruction_type_t type;
66     void* data;
67 } instruction_t;
```

Listing 6 – Structure pour une instruction

Dans ce code, nous avons l'ensemble des structures pour les instructions que nous avons besoin pour le projet.

Chaque structure de données est définie avec des champs spécifique pour stocker les informations qu'ils ont besoin telles que les coordonnées en X et Y, la hauteur etc...

3.7 Structure d'une liste d'instructions

```c
typedef struct instruction_list_node_type {
    instruction_t* instruction;
    struct instruction_list_node_type* next;
} instruction_node_t;

typedef struct instruction_list_type{
    instruction_node_t* head;
    instruction_node_t* tail;
} instruction_list_t;

typedef struct call_procedure_data_type{
    char* procedure_name;
    instruction_list_t* arguments;
} call_procedure_data_t;

typedef struct assignment_data_type{
    char* variable_name;
    instruction_t* value_expression;
} assignment_data_t;

typedef struct conditional_data_type{
    instruction_t* condition_expression;
    instruction_list_t* then_instructions;
    instruction_list_t* else_instructions;
} conditional_data_t;

typedef struct while_loop_data_type{
    instruction_t* condition_expression;
    instruction_list_t* loop_instructions;
} while_loop_data_t;

typedef struct for_loop_data_type{
    symbol_t* variable;
    symbol_t* start;
    symbol_t* end;
    int step;
    instruction_list_t* loop_instructions;
} for_loop_data_t;
```

Listing 7 – Structure pour une liste instruction

La structure *instruction_node_t* contient un pointeur vers une isntruction et un pointeur vers le noeud suivant.

La structure *instruction_list_t* contient un pointeur de la liste d'instruction et un pointeur vers la tête de la liste.

Pour les structures *call_procedure_data_t*, *assignment_data_t*, *conditional_data_t*, *while_loop_data_t* et *for_loop_data_t* contienent les données nécessaires à la bonne exécution du code.

4 Section Lex

4.1 Nos options

```
%option nounput
%option noinput
%option yylineno
```

Listing 8 – Option Lex du projet

Nous avons utilisé trois options pour ce programme, la première est le nounput, la seconde noinput et la dernière yylineo :

La première option utilisée est le noumput. Cette option permet d'éviter la génération d'erreur lorsque l'analyseur lexical rencontre une entrée invalide.

La seconde option est le noinput. Cette option est utile lorsque l'analyseur lexical ne doit pas lire sur l'entrée standard.

La troisiéme option est le yylieno. Cette option permet d'activer le suiveur de ligne dans l'analyseur lexical. Les numéros de lignes sont stockés dans une variable globale.Nous nous en servirons pour indiquer ou est ce qu'il y a une erreur de lecture dans notre fichier.

4.2 Notre message d'erreur

```
void yyerror(const char *msg)
{
    fprintf(stderr, "Erreur ligne %d : %s\n", yylineno, msg);
}
```

Listing 9 – Message d'erreur

La fonction yyerror est une focntion permettant la gestion des erreurs dans notre programme utilisant les analyseurs lexical LexYacc. Cette fonction prend en paramètre un message *msg* qui indique l'erreur rencontrée. En plus, de nous donner l'erreur rencontrée, nous avons ajouter l'affichage de la ligne d'erreur grâce à la variable globale *yylineo*.

4.3 Notre analyseur syntaxique

```
"level"         {return LEVEL; }
"end"           {return END; }

"put"           { return PUT; }
"get"           { return GET; }

"empty"         { return EMPTY_YACC; }
"BLOCK"         { return BLOCK_YACC; }
"TRAP"          { return TRAP_YACC; }
"LIFE"          { return LIFE_YACC; }
"BOMB"          { return BOMB_YACC; }
"DOOR"          { return DOOR_YACC; }
"ENTER"         { return ENTER_YACC; }
"EXIT"          { return EXIT_YACC; }
"LADDER"        { return LADDER_YACC; }
"ROBOT"         { return ROBOT_YACC; }
"PROBE"         { return PROBE_YACC; }
"KEY"           { return KEY_YACC; }
"GATE"          { return GATE_YACC; }

-?[0-9]+        { yylval.value = atoi(yytext); return NUM; }

[a-zA-Z][a-zA-Z0-9]*       { yylval.lettre = strdup(yytext); return IDENTIFIER; }

","             {return VIRG; }

"("             {return PARO; }
")"             {return PARF; }

"+"             { return ADDITION; }
"-"             { return SOUSTRACTION; }
"*"             { return MULTIPLICATION; }
"/"             { return DIVISION; }
"="             { return EGALE; }

\n            { yylineno++; }
[[:space:]]+    {}
.               { fprintf(stderr, "Error: invalid character %s\n", yytext); }
```

Listing 10 – Notre analyseur syntaxe

5 Section Yacc

5.1 Notre Union

```
%union {
        block_t block;
        int value;
        int coordX;
        int coordY;
        char* lettre;
        symbol_t* symbol;
    }
```

Listing 11 – Union Yacc

Le bloc si-dessus est un bloc d'union qui est utilisé pour stocker divers types de données durant l'analyse syntaxique. Nous utiliserons dans notre projet six type de données différentes :

- block_t block : Membre de type block_t vue dans le chapitre précédant.
- int value : Champ de type int qui se nomme value dans notre code Yacc
- int coordX : Champ pour stocker une coordonnée X
- int coordY : Champ pour stocker une coordonnée Y
- char* lettre : Champ de type char* pour stocker des mots ou des lettres
- symbol_t* symbol : Champs permettant le stockage de tous type de symboles.

5.2 Nos Token

```
%token LEVEL END

%token EMPTY_YACC BLOCK_YACC TRAP_YACC LIFE_YACC BOMB_YACC DOOR_YACC ENTER_YACC
EXIT_YACC LADDER_YACC ROBOT_YACC PROBE_YACC KEY_YACC GATE_YACC
%token BLOCK_VAL_YACC

%token GET PUT


%token PARO PARF VIRG NUM PVRIGULE
%token SUP

%token ADDITION SOUSTRACTION DIVISION MULTIPLICATION EGALE SUPEGAL

%token SYMBOLE

%token PRC_YACC LADDER_PRC_YACC RECT_YACC FRECT_YACC HLINE_YACC VLINE_YACC

%token IF_YACC THEN_YACC ELSE_YACC

%token WHILE_YACC DO_YACC FOR_YACC TO_YACC STEP_YACC
```

Listing 12 – Token Yacc

En Yacc, un token est une unité lexical. Dans le code ci-dessus, vous trouverez l'ensemble de nos tokens utilisés dans notre fichier Yacc. Les tokens sont un lien entre le fichier Lex et le fichier Yacc.

5.3 Notre grammaire

```
file: level_file_list
        | instruction_proc_list
        ;

    level_file_list: level_file
                    | level_file_list level_file
                ;

    instruction_proc_list: instruction_proc
                    | instruction_proc_list instruction_proc
                    ;

    instruction_list :  instruction_list instruction
                |   instruction
                ;

    instruction :   instructionPUTNombre
                |   instructionPUTVariable
                |   instructionProcedure
                |   affectation
                |   END {....}
                ;
```

Listing 13 – Régle généraliste pour la bonne lecture d'un fichier

La règle de grammaire nommé *file* possède deux options. La première option a pour but de contruire une liste de fichiers de niveau. Ellle permet de combiner plusieurs niveaux dans un fichier. La seconde option permet de contruire une liste d'intructions procédure.

La règle de grammaire nommée *level_file_list* possède deux options également. La premier est dans le cas où nous avons qu'un seul niveau dans notre fichier. La seconde est écrite dans le cas ou nous avons plusieurs niveaux dans le même fichier.

La règle de grammaire nommée *instruction_proc_list* possèdent deux options aussi pour les même raisons que la grammaire *level_file_list*.

La règle de grammaire *isntruction* possèdent de multiples possibilitées de travail. La première consiste a faire exécuter des option PUT avec des coordonnés X et Y qui sont numériques. La seconde option consiste à remplacer les coordonnés X et Y par des variables (symbole) qui peuvent subire des opérations de calcul entre elles. La troisième sert dans le cas où nous devons poser un bloc sur la map depuis une instruction/procédure définie auparavant. la troisième est une opération d'affection, elle permet d'assigner des valeurs à des variables (symbole). La dernière règle est le TOKEN end pour dire qu'il s'agit de la fin d'un cycle et peut-être la fin du fichier. Les *{...}* signifie qu'il y a des instructions en C dans cette accolade qu'on ne developpera pas dans le rapport.

```
level_file: LEVEL {....} instruction_list ;
```

Listing 14 – Lecture d'un niveau

La règle *level_file* commence par le token LEVEL ce qui marque le début d'un niveau. Nous avons ensuite une suite d'instruction en C. Pouis finir nous faisons appel à une autre grammaire présente plus haute pour pouvoir créer des cycles de lecture du fichier.

```
instruction_proc :

                    PRC_YACC
                    {
                        ...
                    }    LADDER_PROC FOR_LOOP_PROC PUT_PROC END
                |   PRC_YACC{
                        ...
                    }   RECT_PROC FOR_LOOP_PROC PUT_PROC PUT_PROC END
FOR_LOOP_PROC PUT_PROC PUT_PROC END
                |   PRC_YACC{
                        ...
                    }   FRECT_PROC FOR_LOOP_PROC FOR_LOOP_PROC PUT_PROC END END
                |   PRC_YACC{
                        ...
                    }   HLINE_PROC FOR_LOOP_PROC PUT_PROC END
                |   PRC_YACC{
                        ...
                    }   VLINE_PROC FOR_LOOP_PROC PUT_PROC END
                |   END
                {
                    ...
                }
                | level_file
                ;
;
```

Listing 15 – Grammaire pour l'instruction d'une procédure

Nous allons voir à present la règle de production qui se nomme *instructions_proc*. Cette règle est composé de sept membres. Nous allons voir pour commencer les cinqs premiers puis nous verrons les deux dernières par la suite.

Les cinq premières instructions commencent par un symbole non-terminal (PRC_YACC) suivi d'une séquence d'autre symboles non-terminaux et terminaux. Avec les insctructions procédures (PRC_YACC) données dans l'extrait de code ci-dessus, nous pouvons réaliser les procédures donné dans le fichier texte test. Si nous prenons pour exemple la procédure pour poser une échelle, nous pouvons voir que nous aurons une boucle dans laquelle nous auront une instrcution PUT qui nous permettera de la poser sur la map. A la suite de cela nous avons le 'END' qui permet de donner la fin des instruction présente dans la boucles. Nous avons garder cette même réflexion pour les autres procédures.

L'instruction END qui est à la fin nous permet de donner la fin d'une procédure. L'instruction *level_file*, nous permet de passer sur la génération d'un monde à la suite d'un enregistrement de divers procédures dans notre mémoire.

```
LADDER_PROC : LADDER_PRC_YACC PARO affectation VIRG affectation VIRG affectation PARF
              {
                 ...
              }
              ;

 RECT_PROC : RECT_YACC PARO affectation VIRG affectation VIRG affectation VIRG affectation VIRG affectation PARF
              {
                  ...
              };

 FRECT_PROC : FRECT_YACC PARO affectation VIRG affectation VIRG affectation VIRG affectation VIRG affectation PARF
              {
                  ...
              };

 HLINE_PROC : HLINE_YACC PARO affectation VIRG affectation VIRG affectation VIRG affectation PARF
              {
                  ...
              };

 VLINE_PROC : VLINE_YACC PARO affectation VIRG affectation VIRG affectation VIRG affectation PARF
              {
                  ...
              };

 FOR_LOOP_PROC :
              FOR_YACC PARO affectation PVRIGULE affectation SUPEGAL SYMBOLE PVRIGULE SYMBOLE EGALE SYMBOLE ADDITION NUM PARF
              {
                  ...
              }
              | FOR_YACC PARO affectation PVRIGULE affectation SUPEGAL SYMBOLE ADDITION SYMBOLE SOUSTRACTION NUM PVRIGULE SYMBOLE EGALE SYMBOLE ADDITION NUM PARF
              {
                  ...
              }
              | FOR_YACC PARO affectation PVRIGULE affectation SUP affectation ADDITION affectation PVRIGULE SYMBOLE EGALE SYMBOLE ADDITION NUM PARF
              {
                  ...
```

```
39                }
40                ;
41
42    PUT_PROC : PUT PARO SYMBOLE VIRG SYMBOLE VIRG LADDER_YACC PARF
43                {
44                    ...
45                }
46                |
47                PUT PARO SYMBOLE VIRG SYMBOLE VIRG SYMBOLE PARF
48                {
49                    ...
50                }
51                ;
```

Listing 16 – Régle de grammaire pour les différentes procédures

L'ensemble des instructions ci-dessus représente les procédure que nous pouvons rencontrer dans nos fichiers de test. L'ensemble des procédures prennent en entrée et produisent dans le code ... un résultat de sortie comme le stockage des varaibles nécessaires à la bonne exécution de l'ensemble des procédures.

Nous allons vous expliquer rapidement chaque instruction :

— *LADDER_PROC* est une instruction qui permet de créer une échelle en utilisant trois paramètres. Les trois paramètres sont générés depuis la grammaire d'affectation que nous aborderons plus tard.
— *RECT_PROC* est une instuction qui a pour but de créer un rectangle en utilisant cinq paramètres en utilisant la règle de grammaire d'affectation.
— *FRECT_PROC* est une instuction qui a pour but de créer un rectangle plein en utilisant cinq paramètres en utilisant la règle de grammaire d'affectation.
— *HLINE_PROC* est une instuction qui a pour but de créer une ligne horizontale en utilisant trois paramètres en utilisant la règle de grammaire d'affectation.
— *VLINE_PROC* est une instuction qui a pour but de créer une ligne verticale en utilisant trois paramètres en utilisant la règle de grammaire d'affectation.
— *FOR_LOOP_PROC* est une instruction qui permet de créer une boucle for avec plusieurs symboles qui sont directement gérés par affectation. Il y a trois type de boucle spécifiés dans cette grammaire car nous avons les boucles for avec une condition d'arret superieur ou égale ou juste supérieur et celle où nous faisons des sommes de différents symboles pour obtenir une condition d'arret. Les paramètres nécessaires sont : une variable de départ, une variable de fin et un pas d'augmentation d'où le *SYMBOLE EGALE SYMBOLE ADDITION NUM* qui permet de définir le pas.
— *PUT_PROC* Il s'agit d'une instruction qui met en place à l'écran le bloc demandé. Quand il s'agit d'une échelle, nous avons fait une règle spécifique vis à vis d'un problème rencontré avec la règle dans le fichier Lex sur LADDER.

```
instructionPUTNombre :
        PUT PARO expression VIRG
        {
            ...
        }
        expression VIRG
        {
            ...
        }
        instructionBlock PARF
        ;
```

Cette règle de grammaire définit la syntaxe pour une instruction ”PUT” qui met en place un block sur une map level. Nous allons voir chaque partie de cette instruction. Pour la première partie, nous définissions la coordonnée en X de notre bloc. Pour la seconde partie, nous faisons de même mais pour la coordonnée en Y. Dans la dernière ligne de cette instruction, nous allons rechercher les la grammaire instructionBlock qui nous sera détaillée plus tard mais qui permet de placer un block sur la map.

```
instructionPUTVariable :
        PUT PARO SYMBOLE VIRG
        {
            ...
        } expression VIRG
        {
            ...
        }
        instructionBlock PARF
        |
        PUT PARO expression  VIRG
        {
            ...
        } SYMBOLE VIRG
        {
            ...
        }
        instructionBlock PARF
        |
        PUT PARO SYMBOLE  VIRG
        {
            ...
        } SYMBOLE VIRG
        {
            ...
        }
        instructionBlock PARF
        ;
```

La règle de grammaire *instructionPUTVariable* permet de faire passer soit une variable ou un nombre en paramètre pour donner les coordonés d’un bloc. Elle appe-

lera ensuite la règle de grammaire *instructionBlock* en lui passant des coordonnées X et Y. Nous verrons la règle *instructionBlock* dans la suite de ce rapport.

```
affectation :
            SYMBOLE
            {
                ...
            }
            |
            SYMBOLE EGALE NUM
            {
                ...
            }
            |
            SYMBOLE EGALE SYMBOLE
            {
                ...
            }
            |
            SYMBOLE EGALE SYMBOLE ADDITION NUM {
                ...
            }
            |
            SYMBOLE EGALE SYMBOLE ADDITION SYMBOLE {
                ...
            }
            ;
```

Listing 17 – Affectation d'un symbole

La règle de grammaire d'*affectation* consiste à donner une valeur à une varaible. Dans cette grammaire, nous avons plusieurs règles pour spécifier les types d'affectation possible.

La première règle : il s'agit de transformer une lettre en un symbole en mettant la valeur de la lettre à 0. (X) ;

La deuxième règle : a pour but d'affecter un nombre à une variable en utilisant un traitement en langage C adapté (X = 2) ;

La troisième règle : a pour but de faire la somme du contenu d'une variable avec un nombre (X = X + 1) ;

La quatrième règle : a pour but de faire la somme de deux symboles. (X = X + Y) ;

Chaque bloc de règle est suivi par des accolades qui permettent de réaliser les actions à effectuer. Ces actions peuvent être la création d'un symbole dans la table de symbole, la mise à jour d'un symbole en modifiant la variable ou réaliser tout type d'opération nécessaire à la bonne gestion des symboles.

```
    expression :
        | NUM ADDITION NUM  { ... }
        | NUM SOUSTRACTION NUM  { ... }
        | NUM MULTIPLICATION NUM  { ... }
        | NUM DIVISION NUM
        {
            ...
        }
        | '(' NUM ')'   {... }
        | NUM { ... }
        | NUM EGALE NUM { .. }
        ;
```

Listing 18 – Expression de calcul pour les nombres

La régle de grammaire *expression* permet de réaliser des opérations mathématique entre deux nomnbre. Les opérations mathématiques que nous pouvons réaliser sont les additions, les soustractions, les multiplication, les divisions (avec verification des diviseurs) ainsi que comparée si deux nombres sont égales.

```
    instructionBlock :
                        block
                        {
                            ...
                        }
                        |
                        GET PARO NUM VIRG NUM PARF
                        {
                            ...
                        }
                        ;
```

Listing 19 – instruction pour un Block

La règle de grammaire *instructionBlock* permet soit de lancer l'édition d'un bloc sur la map ou soit de montrer le contenu d'un bloc sur un point précis de la map. La première règle de grammaire envoie directement sur la grammaire bloc qu'on verra par la suite *block*. La seconde règle de grammaire a pour but de récupérer les valeurs et de les mettre dans des variables X et Y pour ensuite réaliser une recherche dans notre tableau de Blocs.

```
block: BLOCK_YACC
                        {
                            ...
                        }
        | TRAP_YACC
                        {
                            ...
                        }
        | LIFE_YACC
                        {
```

```
11                              ...
12                          }
13          | BOMB_YACC
14                          {
15                              ...
16                          }
17          | DOOR_YACC PARO expression PARF
18                          {
19                              ...
20                          }
21          | ENTER_YACC
22                          {
23                              ...
24                          }
25          | EXIT_YACC
26                          {
27                              ...
28                          }
29          | LADDER_YACC
30                          {
31                              ...
32                          }
33          | ROBOT_YACC
34                          {
35                              ...
36                          }
37          | PROBE_YACC
38                          {
39                              ...
40                          }
41          | KEY_YACC PARO expression PARF
42                          {
43                              ...
44                          }
45          | GATE_YACC PARO expression PARF
46                          {
47                              ...
48                          }
49                ;
```

Listing 20 – Bloc

Dans cette grammaire, nous pouvons voir que nous avons que le noms des blocks (et pour certains une expression).

Pour tous ceux qui où il y a que le nom du bloc on exécutera directement la méthode pour l'intégrer à la map. Pour ceux qui possèdent une expression permettent d'ajouter une option tel que le numéro de la porte ou le numéro de la clé / Porte.

```
instructionProcedure :
    FRECT_YACC PARO NUM VIRG NUM VIRG NUM VIRG NUM VIRG BLOCK_YACC PARF
    {
        ...
    }
    ;
```

Listing 21 – Bloc

La grammaire sert à lire les appels procédure dans la partie LEVEL ... END

6 Scénarios

Les exemples sont fournis dans un dossier nommé txt.
Voici la liste des fichier de test a essayer pour les codes LexYacc fonctionnel :
— exo1_1A.txt
— exo1_1B.txt
— exo1_1C.txt
— exo1_2A.txt
— exo1_2B.txt
— exo1_2C.txt
— exo1_3A.txt
— exo1_3B.txt
— exo1_3C.txt
— test_level3.txt
— test_level4.txt
Voici la liste des fichier de test à essayer pour les codes LexYacc fonctionnel :
— test_level5.txt

7 Conclusion

Finalement, nous avons pu réaliser un code fonctionnel mais pas complètement fini.

Nous fournissons en plus d'une version final une version test qui vous permet de vous rendre compte de l'avancée actuelle des codes malgré, qu'il ne soit pas fini et non optimisé.

Table des figures

Liste des tableaux