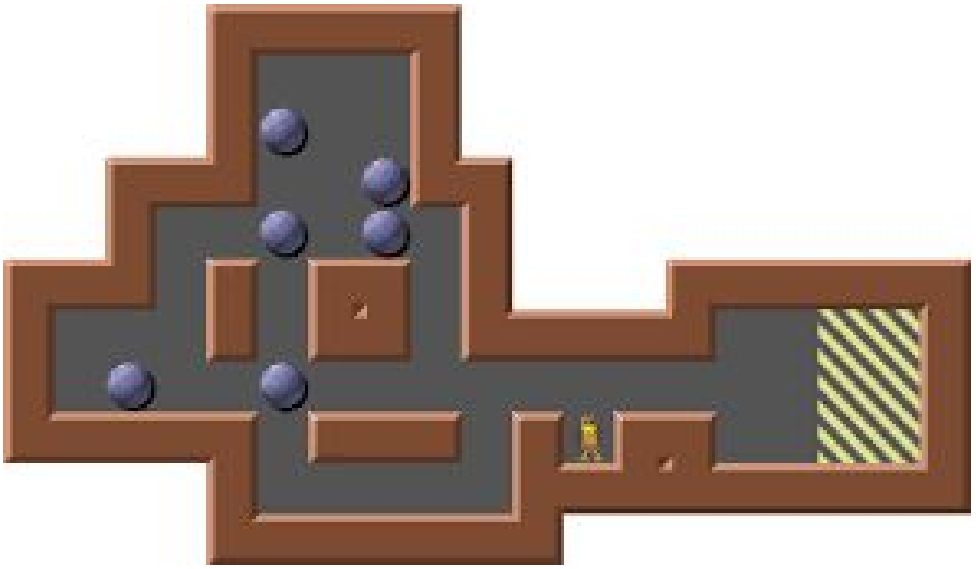
Projet Sokoban

INFO602



GIGOUT Thomas – DAUNIQUE Wilfried

Cyril Rabat

Compte rendu de projet

Sommaire

Introduction

Nous souhaitons réaliser un analyseur syntaxique à l'aide de Lex et Yacc (Flex & Bison). Nous pourrons donc réaliser un ensemble de grammaires et de mots reconnus afin de reconnaître soit un pseudo code, soit un fichier JSON.

La finalité étant un mini-jeu style Sokoban en fonction d'un plateau et d'un algorithme prédéfinis.

Nous souhaitons donc développer un programme en Yacc qui déplace un robot sur un plateau en fonction d'un algorithme définis dans des fichiers externes (respectivement en JSON et en texte) tout deux analysés par Lex&Yacc.

Solution apportée

Comment résoudre ce problème ?

Tout d'abord, nous devons définir dans le fichier Lex l'ensemble de mots que l'on pourra reconnaître dans les différents fichiers lus. Ensuite, nous devons définir dans le fichier Yacc l'ensemble des grammaires possibles avec les mots reconnus. Le but étant d'utiliser la redondance de certains éléments pour remonter les valeurs jusqu'à l'endroit voulu. Le *main* s'occupera ensuite de récupérer ces valeurs, stockées par les grammaires dans une liste, table de hachage ou structure préalablement allouée et déclarée comme variable globale. Ainsi, le *main* pourra accéder aux différents grammaires et mots lus, dans le bon ordre grâce aux listes. Concernant les variables du pseudo code, la table de hachage sera utile pour savoir si une variable a déjà été définie ou non et modifier sa valeur ou "l'allouer".

Le projet se lance donc avec la commande : `./projet plateau.json code.txt`.

Ainsi, nous allons donc ouvrir et lire plateau.json pour le parser avec *yyparse()* pour l'analyser et initialiser le plateau dans le code. Répéter l'opération avec le deuxième argument à savoir "code.txt".

La *main* aura alors tout ce qu'il lui faut pour exécuter le programme.

Lex / Flex - Analyseur lexical

Dans un premier temps, nous allons définir les mots que l'on peut reconnaître. Nous avons donc divisé dans le code en 3 parties. La première partie est les mots que l'on pourra reconnaître dans les deux fichiers, que ce soit du JSON ou du pseudo code. La deuxième partie est les mots reconnus dans le JSON seulement, et la troisième les mots du pseudo code.

Voici la liste des mots reconnus et utilisés :

Global	ENTIER, OPERATEUR
JSON	HAUTEUR, LARGEUR, X, Y, TYPEJSON BLOC BILLE CAISSE TROU CASE CASES DEBUT DIRECTION BAS HAUT DROITE GAUCHE
Pseudo code	TYPE PROC FUNC FINTQ FINPROC FINFUNC NOM F_AVANCE F_DROITE F_GAUCHE INFEG SUPEG EGGEGG INF SUP

La plupart des **définitions** sont des chaînes de caractères prédéfinies du type `HAUTEUR "\"hauteur\""` sauf ENTIER, NOM et OPERATEUR.

```
ENTIER    [0-9]+
OPERATEUR [*+-/]
NOM       [A-Za-z]+
```

La raison pour laquelle nous avons défini tous ces mots reconnus comme tel est que par exemple, pour le JSON, le sujet nous donne une syntaxe bien spécifique. Il ne pourra pas y avoir de plateau avec *height* et *witdh* à la place de *hauteur* et *largeur* par exemple. Les seuls valeurs susceptibles de changer sont donc ENTIER et NOM.

Cette méthode nous facilite la définition d'une grammaire, notamment pour le fichier JSON.

Pour finir, toutes ces définitions de mots, ces **règles**, retournent une chaîne de caractère de la forme :

```
{HAUTEUR}   { return HAUTEUR; }
```

ou

```
{OPERATEUR} { return *yytext; }
```

à l'exception de l'entier défini comme suit :

```
{ENTIER}    { yylval.intval = atoi(yytext); return ENTIER; }
```

Yacc / Bison - Analyseur syntaxique

Nous allons retrouver dans ce fichier plusieurs choses. Premièrement dans les %{}%, les **includes** et les variables globales qui vont servir au stockage des résultats des grammaires analysées. Ainsi, ces résultats pourront être interprétés et utilisés dans le *main*. La partie des **définitions** sert à définir tout ce que l'on va utiliser dans les grammaires, notamment les noms des grammaires et ce qu'elles retournent si elles retournent quelque chose.

Ici nous avons donc une %union qui permet de définir deux types de retours possibles pour les grammaires : *intval* et *string*.

Majuscule : définitions Lex Minuscules : Noms de grammaires Yacc

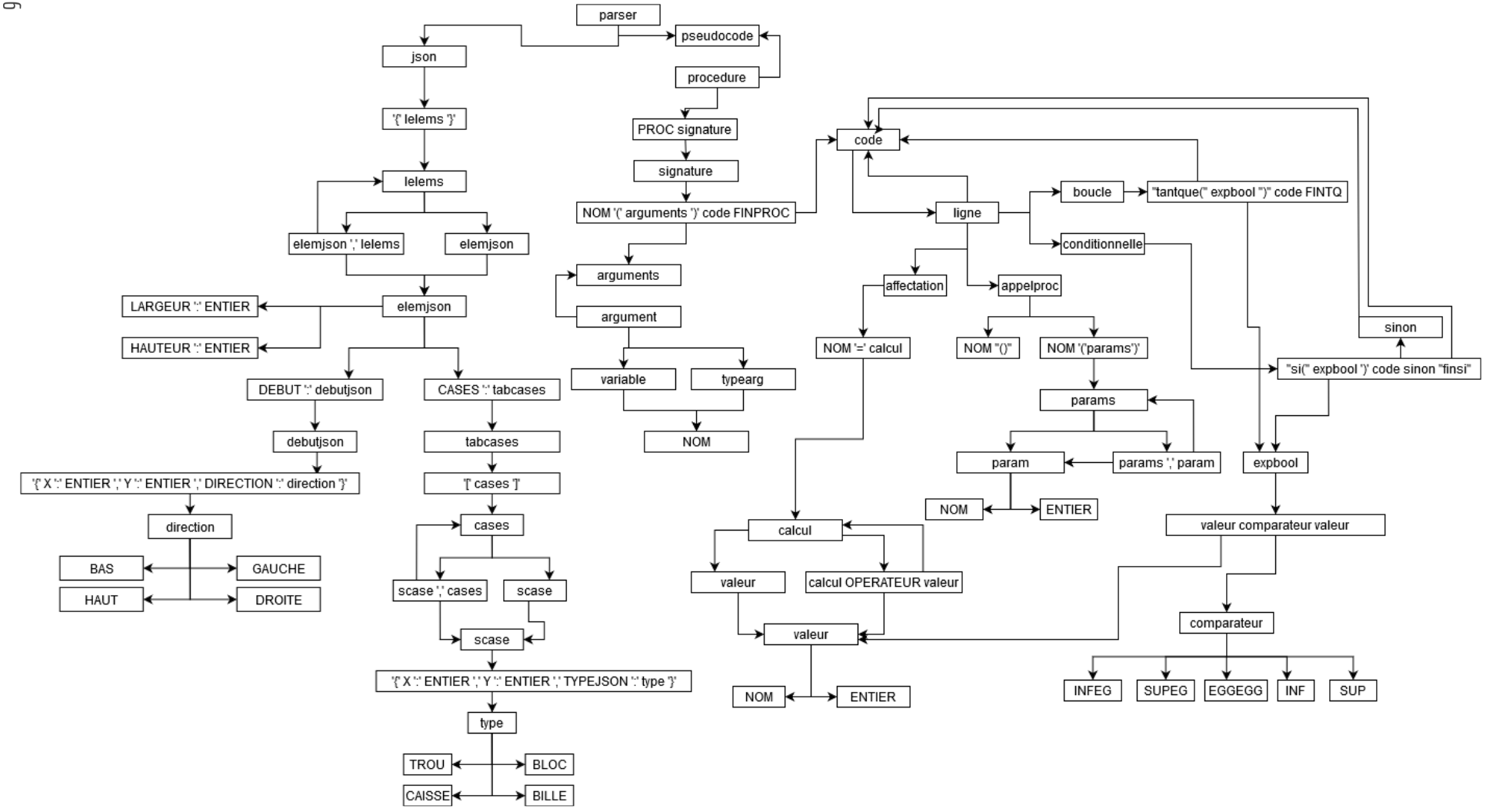
%token	VRAI FAUX HAUTEUR LARGEUR X Y TYPEJSON BLOC BILLE CAISSE TROU CASE CASES DEBUT DIRECTION BAS HAUT DROITE GAUCHE TYPE PROC FUNC F_AVANCE F_DROITE F_GAUCHE FINTQ FINPROC FINFUNC
%token <intval>	ENTIER direction type
%token <string>	NOM OPERATEUR INFEG SUPEG EGGEGG INF SUP comparateur valeur ligne expbool

Nous avons ensuite les **règles** qui sont les grammaires et ce qu'elles font. Pour résumer, voici un arbre représentant le cheminement des grammaires :

(cf. page suivante)

Nous avons choisi d'utiliser cette modélisation afin de remonter les informations importantes au bon endroit, et pouvoir envoyer aux variables globales les choses que l'on a besoin dès qu'elles sont reconnues.

De plus, la grammaire ***parser*** détermine si on lit un fichier au format json ou un fichier autre (ici forcément pseudo-code) grâce aux grammaires du même nom : ***json*** et ***pseudocode***. Ainsi, même si l'on fait deux yyparse() dans le main, on ne parcourra pas toutes les grammaires dans le vide, on utilisera uniquement celles qui correspondent au format du fichier.


parser
pseudocode
json
procedure
'{' lelems '}'
PROC signature
lelems
signature
elemjson ',' lelems
elemjson
NOM '(' arguments ')' code FINPROC
arguments
elemjson
LARGEUR ':' ENTIER
HAUTEUR ':' ENTIER
argument
DEBUT ':' debutjson
CASES ':' tabcases
variable
typearg
debutjson
tabcases
NOM
'{' X ':' ENTIER ',' Y ':' ENTIER ',' DIRECTION ':' direction '}'
'[' cases ']'
direction
cases
BAS
GAUCHE
HAUT
DROITE
scase ',' cases
scase
scase
'{' X ':' ENTIER ',' Y ':' ENTIER ',' TYPEJSON ':' type '}'
type
TROU
BLOC
CAISSE
BILLE
code
ligne
boucle
"tantque(" expbool ")" code FINTQ
conditionnelle
affectation
appelproc
sinon
NOM '=' calcul
NOM "()"
NOM '('params')'
"si(" expbool ')' code sinon "finsi"
params
param
params ',' param
expbool
NOM
ENTIER
calcul
valeur comparateur valeur
valeur
calcul OPERATEUR valeur
comparateur
valeur
NOM
ENTIER
INFEG
SUPEG
EGGEGG
INF
SUP

Grammaires

Idéalement, les grammaires sont décomposés -> parser: json | pseudocode. Ainsi on peut savoir dès qu'on parse le fichier vers quelles autres grammaires s'orienter.

JSON

Nous avons donc une grammaire json qui peut être composée d'une liste d'éléments *lelems* compris entre accolades. Pour définir un lelems, nous définissons qu'il peut être un *elemjson* ',' *lelems* **ou** un *elemjson* pour la récursivité et ainsi autoriser le fait d'avoir un ou plusieurs *elemjson*. Bien entendu il en faut obligatoirement quatres du schéma donné par le sujet pour que l'exécution fonctionne correctement.

elemjson

D'où le fait qu'un *elemjson* puisse être soit :

- LARGEUR ':' ENTIER où LARGEUR = "largeur"
- HAUTEUR ':' ENTIER où HAUTEUR = "hauteur"
- DEBUT ":" *debutjson* où DEBUT = "debut"
- CASES ":" *tabcases* où CASES = "cases"

Lorsque l'on reconnaît soit largeur soit hauteur, on place respectivement les valeurs ENTIER trouvées dans plateau->largeur ou plateau->hauteur où plateau est la variable globale représentant le plateau de jeu.

debutjson

Un *debutjson* définit le début du robot, à savoir sa position (x,y) et sa direction.

debutjson:

'{'X':'ENTIER','Y':'ENTIER','DIRECTION':'*direction*'}'

direction: BAS|HAUT|DROITE|GAUCHE (Chacune des valeurs renvoie une constante définie telle que M_BAS M_HAUT ... pour envoyer un entier représentant la direction.

Ainsi, on pourra faire la même manipulation qu'avec largeur et hauteur, mais dans la variable *robot_t robot*, elle aussi globale. Nous allons aussi ajouter le fait qu'un robot est placé à cette case grâce à la liste de cases de type *liste_t* qui nous permettra par la suite de savoir si une case est déjà occupée ou non.

tabcases

Le *tabcases* est défini par '[' *cases* ']' ou *cases* est aussi une grammaire récursive afin de savoir s'il y en a plusieurs ou une seule.

cases

cases: scase ',' *cases* | *scase{};*

On va donc parcourir toutes les cases et les placer, comme pour le robot, dans la liste_t qui représente les cases du jeu. Ainsi, chaque case est définie comme telle *'{'X':'ENTIER','Y':'ENTIER','TYPEJSON':'type'}'* où *type* peut, de la même façon que la direction que le robot, peut valoir M_CAISSE, M_TROU etc...

Pseudo-code

Le pseudo code est directement récursif, car un pseudo code est une suite de procédures si on y réfléchit. Chaque procédure contient du code avec des instructions, des lignes.

C'est pourquoi on définit le pseudocode ainsi :

pseudocode

procedure pseudocode{}

|

procedure{};

On peut maintenant définir qu'une *procedure* est soit une fonction utilisable dans le *main*, soit le *main* en lui même

procedure : PROC signature;

signature

signature : NOM '(' arguments ')' code FINPROC;

Ici on peut s'apercevoir qu'une procédure peut ne pas avoir d'arguments. D'où la récursivité de

arguments

argument: (*argument* | *arguments ',' argument;*)

De même, le *code* est récursif : *ligne code | ligne*;

ligne

On définit donc une ligne en quatres possibilités :

- affectation
- appel de procédure
- conditionnelle (on utilise que si sinon finsi)
- boucle (on utilise que tantque)

Cela nous donne :

ligne: affectation{} | appelproc{} | conditionnelle{} | boucle{};

Une *affectation* est donc un NOM de variable suivi d'un '=' et d'un *calcul*. Le *calcul* est lui aussi récursif, car il peut être une seule variable, une seule valeur entière ou un calcul complet avec addition etc.

L'appel de procédure contient ou non des paramètres, d'où la récursivité de *params*.

conditionnelle

La conditionnelle est intéressante car on test si la chaîne trouvée ainsi :

"si(" expbool')' code sinon "finsi"

Nous avons vu *code* plus haut, et le *sinon* est soit vide, ou contient du *code*. Par contre, *expbool* doit comparer deux *valeur* avec un *comparateur*. Le *comparateur* renvoie donc à la manière des éléments JSON précédemment vu soit INFEG (<=), SUPEG(>=), EGGEGG(==) (easter egg), INF(<) ou SUP(>).

Exécution / Affichage

Premièrement nous initialisons évidemment toutes les variables que l'on va manipuler. Les plus importantes sont le plateau, le robot, les "file descriptor" et l'initialisation de la liste_cases avec init_liste().

Nous ouvrons donc le premier fichier passé en paramètre du programme et vérifions s'il existe. Si oui, nous plaçons son FD (file descriptor) dans le *yyin* et nous parsons avec *yyparse();*

Nous sommes censés répéter l'opération une deuxième fois mais nous ne le faisons pas car nous avons rencontré quelques problèmes lors du parse du pseudocode qui seront expliqués plus bas.

Une fois le fichier json parsé, nous pouvons utiliser les valeurs qu'il a rempli dans le plateau pour initialiser comme il faut ncurses et les fenêtres.

L'affichage du plateau rempli fonctionne complètement. Une légende est associée afin de ne pas se perdre dans les couleurs.

Malheureusement, c'est tout ce que nous faisons à l'issu de ce projet.

Difficultés rencontrées

Les difficultés rencontrées ont été principalement lors des définitions de grammaires récursives. Notamment car nous n'avons trouvé aucun moyen viable de débugger les grammaires et ainsi corriger rapidement les éventuelles erreurs. De plus, la visualisation du projet en amont fût assez difficile du fait du peu d'expérience que l'on avait dans l'écriture en Lex&Yacc.

La partie qui nous a causé du tort est principalement la récurisivité des arguments lors d'un appel de fonction. Une erreur sans explications est apparue. Nous pensons à un problème de définition de mot dans Lex. Nous nous sommes entêtés à régler le soucis et avons buté dessus pendant quelques heures sans pour autant régler le soucis. Dès que l'on retiré un $1 dans un sprintf, cela générait un "core dumped". Nous avons tenté de régler le soucis en examinant avec l'outil valgrind, mais nous n'avons pas réussi à résoudre le problème. Si vous souhaitez voir le parsing du pseudo code, vous pouvez décommenter la partie dans le main qui ouvre le deuxième argument et qui le parse.

Nous arrivons à lire quasiment tout le fichier, il manque la dernière ligne où il génère une erreur de syntaxe sur "i".

Nous avons donc perdu énormément de temps sur ces choses là, ainsi que de la motivation. Nous n'avons donc pas eu le temps d'implémenter ce que nous voulions.

Implémentation manquante

Il manque donc les listes chaînées avec la table de hachage même si les structures et les fonctions sont écrites. Voici ce que nous avions prévu :

à l'image de la liste_t pour le json, nous voulions faire une liste_hachage_t et une liste_variables qui permettrait de stocker les instructions de type *affectation* et de faire en sorte de pas les déclarer plusieurs fois si elles l'étaient déjà. Une liste_instructions aurait été implémentée aussi afin de stocker les appels de procédure dans l'ordre de lecture, et ainsi les ressortir dans un ordre FIFO pour pouvoir garantir le bon déroulement de l'algorithme du pseudo-code. Malheureusement, cette histoire de *params* nous a freiné dans cette démarche.

Conclusion

Nous avons réussi la moitié du travail selon nous, à savoir parser un document sur deux. L'affichage avec ncurses était une formalité. Cependant, il ne nous restait pas énormément de choses à faire pour mener à bien ce projet étant donné que les structures sont écrites, et qu'il ne restait plus qu'à stocker ce qu'il fallait grâce aux grammaires si elles fonctionnaient toutes.

Néanmoins, nous avons tout de même beaucoup appris avec ce projet car au début, nous étions constamment en train de chercher dans le cours et sur internet pour la moindre chose, notamment comment organiser le code Lex&Yacc car ce n'était pas du tout intuitif au vu des autres langages que nous voyons en Licence Informatique.