Rapport Projet

INFO0602

Générateur de mondes pour jeu de plateforme

LEVALET Corentin et COLLIGNON Alexandre

2022-2023 Licence 3 INFO S6

Bonne lecture 🙂

Sommaire

Présentation

Ce rapport présente les différentes possibilités de notre travail effectué sur le projet d'INFO0602 utilisant Lex & Yacc. Le projet consiste en un exécutable pouvant prendre en paramètre un nom de fichier ou bien interpréter peu à peu les lignes écrites par un utilisateur.

Pour commencer ce rapport, nous allons dans un premier temps nous pencher sur une première partie expliquant globalement les structures utilisées pour ce générateur ainsi que leurs cas d'utilisation. Dans une seconde partie, nous analyserons toutes les capacités de notre interpréteur en détail en nous penchant sur les règles de la grammaire et comment cela peut être utilisé.

Finalement, pour conclure ce rapport, nous verrons les difficultés rencontrées durant ce projet ainsi que ce qui n'a pas pu être traité à temps selon les instructions initiales.

Structuration

Afin que notre projet soit fonctionnel, nous avons structuré un dossier (celui du dépôt) contenant nos clés maîtres que nous allons expliquer dans chacune des sous-parties suivantes.

Exécutable

Makefile

Le makefile est fait pour créer un exécutable nommé “_projet” à partir des autres objets du projet. La commande “make clean” permet de supprimer tous les fichiers objets .o ainsi que les fichiers générés par Lex & Yacc. De ce fait, la commande “make depend” qui génère les dépendances va tout simplement rater si un make clean a été fait avant puisque le projet dépend de ces fichiers générés. La commande “make all” ou “make” permet de compiler le projet à partir des dépendances dont la génération des fichiers .c de Lex & Yacc.

Il faut donc toujours que le make depend soit fait avant un clean pour éviter les erreurs comme quoi certains fichiers sont introuvables.

_projet.c

Ce fichier est le fichier principal de notre projet, il peut prendre en compte un paramètre pour un fichier d’entrée (optionnel), dans ce cas au lieu de prendre en entrée le clavier de l’utilisateur, il va lire le fichier précisé. Pour lancer l’exécutable avec un fichier, vous avez à disposition dans le dossier “demo/” des fichiers de démonstration (au nombre de 3) que vous pouvez appeler de la manière suivante :
> ./_projet “demo/1_arithmetique.txt”.

Si l’exécutable a reçu un fichier en entrée, il va préparer un fichier en sortie du même nom mais avec “.out” ajouté à la fin. Fichier qui est le résultat d’un niveau plus simple à lire pour un humain, mais qui peut être réinterprété par le projet. On aurait pu faire un fichier binaire du niveau mais nous n’avons pas choisi cette option car c’est “difficile” de vérifier le résultat du fichier et puis que le level n’est pas le même que ce que nous avons pour le projet d’INFO0601/INFO0604.

L’exécutable s’occupe d’initialiser la table des symboles et certaines variables globales utiles pour les fichiers Lex & Yacc.

Fichiers de support

Définition d'un level

Une structure définissant un niveau nous a été donnée pour le projet. Elle possédait déjà quelques fonctionnalités, dont la création d'un niveau (dont le placement de différents types de blocs), et son affichage. Cependant, il y manquait plusieurs choses donc nous avons modifié ces fichiers. Tout d'abord, nous avons rajouté une matrice de blocs afin de pouvoir récupérer efficacement un bloc à des coordonnées (utile pour la fonction get(x,y)). Nous avons changé le type d'un block_t d'un entier vers un enum pour chaque type dans le but de gagner en lisibilité. Enfin, nous avons ajouté une fonction permettant de convertir un niveau en liste d'instructions interprétables par notre projet sous la forme d'un texte. (Fonction utilisée pour générer le fichier .out précédemment expliqué)

Table des symboles et symbole

Une table des symboles est utilisée pour stocker des variables tout au long de l'interprétation. Elle est structurée par une table de hachage de taille définie avec chacune des cases de cette table étant une liste chaînée de symboles. De ce fait, nous avons des fonctions pour déterminer la position d'un symbole dans la table, en ajouter, en rechercher, en supprimer.

Un symbole est défini par son type (dans notre cas seulement un entier par manque de temps, sinon on aurait pu implémenter le type d'un bloc et autres), son nom (par exemple 'x' ou 'chuck_norris'), sa valeur ainsi que sa profondeur afin de savoir à quel niveau la variable a été créée pour pouvoir la supprimer lorsqu'on remonte d'un niveau grâce à une fonction dédiée. (Même fonctionnement que les variables "locales" à des fonctions).

La table des symboles est supprimée et correctement vidée lors de la fermeture de l'interpréteur.

Arbre d'expressions

L'arbre d'expressions n'a pas été utilisé lors du projet par manque de temps mais nous souhaitons quand même énumérer le fait que nous avons créé la structure et les fonctions appropriées.

Capacités et analyse

Partie Lex

Le fichier Lex a pour but de reconnaître les mots-clés et en faire part à Yacc. Le notre reconnaît donc le début d'un “level”, sa fin (“end”), les types de blocs (Robot, probe, empty, etc.), des noms de variables, et des opérateurs. Sans parler de la gestion des espaces et des passages à la ligne et une fonction pour gérer les erreurs sur certaines lignes.

Noms de variables

Lex s’occupe de reconnaître des noms de variables commençant par une lettre minuscule ou majuscule puis un nombre infini de lettres et d’underscore (‘_’). Lorsqu’un nom de variable est reconnu, une recherche dans la table des symboles est faite. Si le symbole est trouvé, on remplit une variable yylval contenant un symbole et on précise à Yacc que c’est une “variable”. Si le symbole n’est pas trouvé, on le crée, on l’ajoute à la table des symboles, puis on remplit la variable yylval avec le nom du symbole.

Partie Yacc

Nous allons ci-dessous expliciter les différentes règles du fichier Yacc.

Les ASSIGNEMENTS

Les règles d’assignements permettent d’ajouter un symbole et/ou de mettre à jour la valeur d’une variable dans la table des symboles.
Il y a donc une règle par type d’opération logique :

- Le += (addition avec auto réaffectation).
- Le ++ (incrémentation).
- Le -- (décrémentation).
- Le -= (soustraction avec auto réaffectation).
- Le + pour l’addition de deux expressions ou/et valeurs.
- Le - pour la soustraction de deux expressions ou/et valeurs.
- Le * pour la multiplication de deux expressions ou/et valeurs.
- Le / pour la division de deux expressions ou/et valeurs.
- Le % pour le reste de deux expressions ou/et valeurs.
- ...

Pour conclure toutes les opérations possibles en C si on oublie celles du décalage de bit.

Les EXPRESSIONS

Les règles d'expression permettent les différentes opérations élémentaires : / + * - % ainsi que la priorité des parenthèses avec des nombres et des variables.

Dans le cas de la division et du reste, nous avons mis un garde-fou permettant la détection des divisions par zéro.

Dans le cas de l'utilisation d'une variable, nous cherchons sa valeur dans la table des symboles et si celle-ci n'est pas trouvée, une erreur est levée.

Les Items

Ces règles permettent de reconnaître les différents objets de la carte. Dans le cas d'objet possèdent des arguments tels que les gates, les keys ou bien les portes, nous vérifions que la valeur passée est cohérente, soit donc une clef entre 1 et 4 (compris), une porte entre 1 et 99 et une gate entre 1 et 4 (compris).

Dans le cadre des arguments étant tous des arguments numéraires, nous avons en conséquence utilisé EXPRESSION pour permettre le support du calcul sur ses valeurs.

GET_FUNC

Il s'agit de la règle permettant de reconnaître la fonction get(x,y) donnant la capacité de retourner l'élément dessiné à cette position. Celle-ci se base sur une expression pour obtenir les arguments x et y rendant, ces arguments calculables.

Pour éviter que les entrées de x et y soient hors de la taille du niveau, une vérification des coordonnées entrées est effectuée.

PUT_FUNC

À l'inverse de la règle GET_FUNC cette règle permet de définir un élément sur la map. Elle prend 3 arguments, les deux premiers sont des EXPRESSION et le dernier est un ITEM.

Nous vérifions le range des coordonnées entrées et nous utilisons des conditionnelles (Un switch case) pour appeler les fonctions C adéquates à l'ajout du bon item sur la carte de jeu.

CONTENT_LIST et CONTENT

Ces règles-ci permettent d'effectuer la liaison entre des PUT_FUNC successives et/ou des ASSIGNEMENT.

ENTER_LEVEL_BLOCK

Cette règle permet d'incrémenter correctement la profondeur des blocs d'instruction et d'initialiser la variable du nouveau niveau. Elle ne reconnaît aucun caractère, elle a juste été créée pour la lisibilité.

LEVEL_BLOCK

Cette règle permet de reconnaître un bloc complet de définition d'un niveau. Ce bloc est composé du token "level" suivit de l'expression ENTER_LEVEL_BLOCK et d'une expression CONTENT_LIST défini plus haut. Cette règle déclenche l'affichage du niveau (une fois construit), déclenche l'enregistrement dans un fichier d'output si celui-ci a été renseigné et fini par libérer de la table des symboles tous les symboles définis au niveau de ce bloc (suivant la profondeur : depth).

La règle se termine par la décrémentation de la profondeur (depth) et l'impression d'un message de succès à l'écran.

GLOBAL_CONTENT et GLOBAL_LIST

Ces règles permettent d'effectuer la liaison entre plusieurs blocs de niveau défini à la suite ainsi que des ASSIGNEMENT qui seraient utilisés en "variable globale".

EXEC

Cette règle est la règle de base permettant de reconnaître une GLOBAL_LIST. Lors de son exécution, nous affichons juste un message confirmant le bon déroulé de la reconnaissance d'entrée, il est en soit optionnel car le programme peut démarrer à partir de GLOBAL_LIST.

Nos difficultés

Parties non implémentées.

Nous avons durant les TPs eu des problèmes de compréhension sur les arbres d'évaluation, Ces problèmes se sont dissipés par la tenue des TDs qui ont suivi, mais nous n'avons malheureusement pas été capables de rattraper notre retard pris. Ainsi, nous n'avons pas pu implémenter les fonctionnalités suivantes :

- L'arbre d'évaluation des expressions
- Les procédures et fonctions
- Les conditions booléennes. (IF, true or false)
- Les boucles (while et for)