Rapport de projet

TABLE DES MATIÈRES

1. Introduction

LaTeX est un langage et un système de composition de documents. Il regroupe une collection de commandes destinées à faciliter l'utilisation du « processeur de texte ». Le programmeur a aussi la possibilité de créer ses propres commandes.

Le sujet du projet est d'écrire en d'introduire deux programmes qui illustrent les commandes **\ASIPRO** et **\SIPRO** en Latex.

Le programme **algo2asm** qui implémente la commande ASIPRO prend en argument un fichier Latex avec l'extension **.tex** contenant la description d'une fonction f dans la syntaxe du paquetage ALgo. Elle traduit la description de cet algorithme en ASIPRO, en produisant un fichier **.asm** correspondant au .tex.

Pour simplifier, il est dit dans le sujet qu'on supposera que le nom du fichier est le même que la fonction décrite dans le fichier (dans l'exemple précédent, on suppose donc que le fichier puissance.tex contient la description ALgo d'une fonction nommée puissance).

Le programme **run** qui implémente la commande SIPRO prend en argument une fonction avec ses arguments s'il y en a. Ensuite, il génère un code assembleur exécutable avec sipro qui calcule la valeur de la fonction.

Ce projet a pour but de mettre en pratique nos connaissances et compétences acquises en cours et travaux pratiques de compilation.

2. Phase Développement

Durant la phase de développement, j'ai utilisé deux outils d'analyse de texte qui sont Flex et Bison. J'ai appris à utiliser ces outils en travaux pratiques. Flex est un analyseur lexical et Bison un analyseur grammatical. Ces deux outils m'ont permis de mettre en place ces deux types d'analyse pour réaliser la commande \ASIPRO.

a. Analyse lexical

L'analyseur lexical examine un flux d'entrée standard dont le fichier porte l'extension .l ou .lex et produit un automate en sortie ayant typiquement le nom de fichier yy.lex.c. Cette analyse se base sur un fichier de configuration qui traduit les expressions rationnelles en automates.

Maintenant, parlons de la configuration de flex qui se trouve dans le fichier **algo2asm.l** à la racine du projet.

Un fichier Flex s'organise en quatre parties. La première étant du code C qui sert à inclure les bibliothèques et autres fichiers nécessaires au bon fonctionnement du programme. Après, il y a la deuxième partie qui comprend les définitions Flex. En troisième partie, la liste des expressions régulières ayant pour but de construire l'automate. Et enfin, en quatrième partie, du code C dont le plus souvent sert à redéfinir la fonction main ou ajouter d'autres fonctions.

Donc dans notre cas, la première partie a permis d'inclure les fichiers nécessaires au bon fonctionnement de l'analyse lexicale.

Après, nous avons les définitions Flex. Dans un fichier LaTex, les commandes commencent par un anti-slash **\,** ainsi j'ai décidé de créer une définition Flex nommée KEYWORD_PREFIX qui se place avant chaque expression rationnelle d'une commande.

Ensuite, on retrouve de nouvelles définitions pour chaque mot clé signifiant une action spécifique (begin, end, set, if, else, dowhile, od, return, incr, decr, dofori, com).

- KEYWORD_BEGIN_COMMAND : début de l'algorithme de la fonction
- KEYWORD_END_COMMAND : fin de l'algorithme de la fonction
- KEYWORD_SET_COMMAND : affectation d'une variable
- KEYWORD_IF_COMMAND : condition si
- KEYWORD_ELSE_COMMAND : condition sinon
- KEYWORD_DOWHILE_COMMAND : boucle tant que
- KEYWORD_OD_COMMAND : fin de boucle tant que et de la boucle pour
- KEYWORD_RETURN_COMMAND : retour d'un résultat d'une fonction
- KEYWORD_INCR_COMMAND : incrémentation d'une variable
- KEYWORD_DECR_COMMAND : décrementation d'une variable
- KEYWORD_DOFORI_COMMAND : boucle pour
- KEYWORD_COM_COMMAND : commentaire

L'ordre de ces définitions représente l'ordre chronologique des fonctionnalités implémentées.

Après ces définitions, il y a l'option noyywrap que je souhaite expliquer car je la trouve importante. Elle me semble importante car elle permet d'indiquer à Flex de ne pas appeler la fonction yywrap() en fin d'analyse du flot d'entrée et d'analyser qu' un fichier.

Par la suite dans cette troisième partie, nous avons la liste des expressions régulières qui utilisent les définitions citées ci-dessus et de nouvelles regex. Chaque expression régulière (regex) retourne un terminal qui sera utilisé dans bison pour comprendre une règle.

Cependant noté, que certaines expressions régulières retournent plusieurs terminaux comme c'est le cas pour le return NUMBER et return IDENTIFIER. En effet, car un nombre peut être composé de plusieurs chiffres et de même qu'un identifiant de variable ou de fonctions peut avoir des minuscules, des majuscules, des chiffres et un tiret du bas.

Un autre point important est que certains retours possèdent un nom inconnu de Flex comme c'est le cas pour IDENTIFIER, NUMBER, AND, OR, EQ, NED, BEGIN_COMMAND ect. C'est tout à fait normal car ils sont créés dans bison et ont une fonction particulière. Nous pouvons utiliser ces nom inconnu car en réalité ils sont connus pour Flex car dans la première partie, nous avons inclus un fichier nommée algo2asm.tab.h qui contient les définitions de jetons pour les terminaux créer par bison.

Pour finir, il y n'y a pas de fonction main car je n'ai pas trouvé l'utilité de définir ma propre fonction. Par conséquent, j'utilise celle de la bibliothèque flex qui exécute cette instruction while (yylex()); et donc pour l'utiliser j'ai préciser dans mon makefile l'option -lfl.

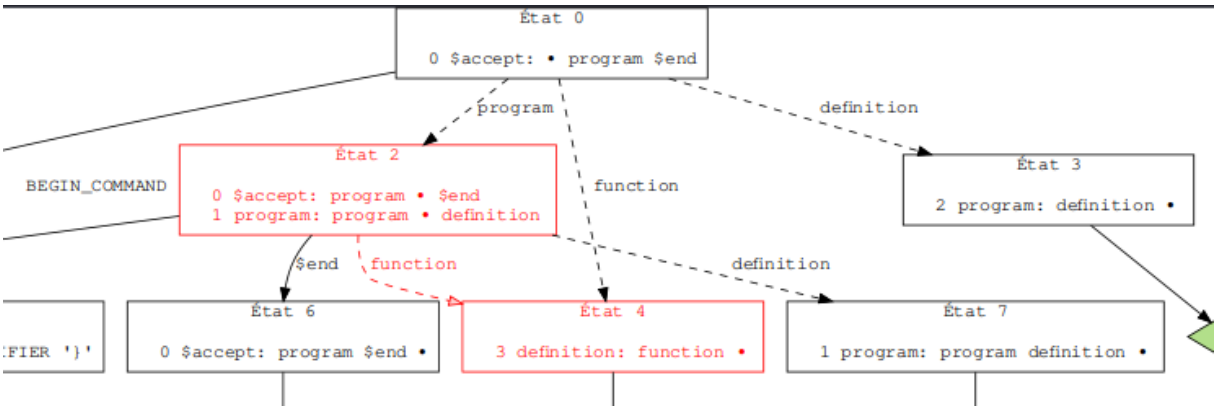
b. Analyse Grammaticale

Le langage d'une grammaire est l'ensemble des mots sur ses terminaux qu'on obtient à partir de l'axiome.

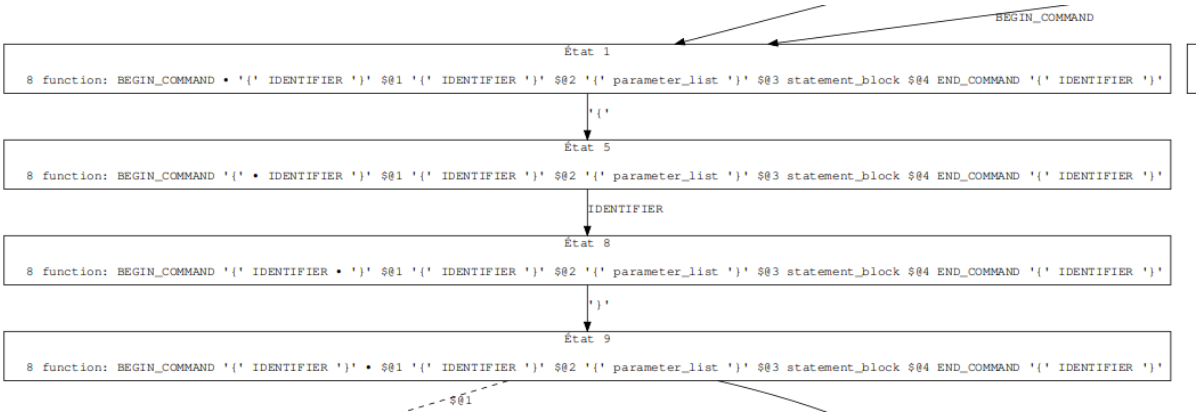
L'analyse grammaticale consiste à construire un analyseur à partir d'une description d'une grammaire contenue dans un fichier un texte. C'est le rôle de l'outil Bison. Il analyse un fichier portant l'extension .y.

À propos de cette analyse, le fichier la concernant est **algo2asm.y** situé à la racine du projet.

Grâce à cette outil, nous pouvons visualiser la grammaire grâce à un graphe comme le montre les copies d'écran ci-dessous :



Copie d'écran de l'axiome du graphe de la grammaire



Copie d'écran des premiers états du de la grammaire

Le graphe nous permet de visualiser tous les états et on en compte plus d'une centaine.

i. Fonctionnalités

Avant de dérouler la liste des fonctionnalités, je tiens à définir quelques points primordiaux pour comprendre la liste des fonctionnalités.

Le premier point est que j'ai utilisé une table des symboles exprimée sous forme d'une liste chaînée dont l'élément le plus récent se trouve en début de file.
Concernant l'implémentation, elle est représentée par une structure C nommée symbol table entry. Celle-ci associe à chaque symbole (nom de variable et fonction) ses informations. Ainsi on y retrouve le nom de la variable, le type de la variable, sa portée (numéro du bloc où elle a été fixé), le nombre de paramètres (pour une fonction), le numéro de variable locales (pour une fonction), le type de retour (pour une fonction) et le symbole suivant. De plus, il y a plusieurs actions sur la table des symboles. On peut chercher un symbole pour savoir s'il existe, ajouter un nouveau symbole et le libérer de la table des symboles.

Le deuxième point concerne les expressions. Chaque expression qui possède un opérateur binaire est associative à gauche et triée en fonction de leurs priorités. La priorité la plus forte revient à la multiplication, la division et le modulo, c'est pourquoi ils sont tout en bas dans la déclaration des priorités.

Programme :

Un programme se constitue d'une définition qui est une fonction ainsi j'ai matérialisé par la 1ère règle : **program : program definition | definition ;**
Cette règle définit ce qu'est un fichier LaTeX. Vous pouvez remarquer qu' un programme peut être constitué de plusieurs définitions, j'ai gardé cela parce que un programme peut évoluer et cette règle ne me semble pas pertinente à enlever.

Fonction :

Une fonction est une définition avec un mot clé début puis un nom de paquetage, suivi d'un nom de fonction avec ou sans paramètre(s). Ensuite, il y a un bloc d'instruction(s) et se termine par le mot clé de fin avec le nom du paquetage identique à celui du début.

Chaque nom représentant un symbole, j'ai choisi de les représenter par un jeton (une unité lexicale) IDENTIFIER. Chaque identifiant étant un symbole est stocké dans la table des symboles.

Argument de fonction :

Un argument de fonction est un symbole enregistré dans la table des symboles référençant une variable locale, de type entière, avec comme numéro de porté égale à celui du premier numéro de la fonction car c'est le début de la fonction.

Ensuite, lorsqu'un nouveau paramètre est ajouté, le compte de nombre de paramètres de la fonction augmente.

Quand l'analyse est terminée à la fin de la fonction, les identifiants de la table des symboles correspondant au paramètres de la fonction sont libérés.

Dans le code assembleur, les arguments de fonction sont récupérés du plus récent au plus ancien.

Bloc d'instruction :

Un bloc d'instruction est composé d'une ou plusieurs instructions. Celui-ci dispose d'un numéro de portée qui est incrémenté à chaque de bloc et décrémenté lorsque chaque bloc se termine. Quand toutes les instructions sont terminées, il y a la libération des symboles appartenant au bloc d'instruction.

Retour de la fonction :

Au moment du retour de la fonction, la valeur se trouve en sommet de pile. Par conséquent, j'ai choisi la méthode indiqué dans l'exemple de la factorielle qui est de dépiler cette expression et la mettre dans le registre ax par convention. Au retour de la fonction, le résultat est mémorisé dans le registre ax.

Condition IF :

La condition IF est une instruction qui peut contenir d'autres d'instructions IF ou autres. C'est pourquoi j'ai décidé de créer un tableau qui indexe la suite de IF avec son propre numéro unique. Les étiquettes else, endif associées au if code assembleur est alors doté de ce numéro correspondant à l'instruction du IF.

À noter que l'étiquette else est présente, même si il y a n'a pas de ELSE car la règle ne différencie pas le IF et le IF ELSE. Mais, elle dit juste que le ELSE peut être vide d'où le %empty.

Concernant le test de l'expression, on dépile la valeur de l'expression et on la compare à 0. Si l'expression est fausse alors alors il y a un saut à l'étiquette de fin du IF. Sinon, il s'ensuit l'exécution du code généré par le bloc d'instruction.

Condition IF ELSE :

La condition IF ELSE est légèrement différente par rapport au IF sans ELSE. La seule différence est que lorsqu'il y a le ELSE, l'étiquette else qui contient le code assembleur du nouveau bloc d'instruction est à exécuter. Ainsi, si la condition est fausse (le code assembleur de l'expression différent de 1) alors c'est le bloc du else qui est exécuté.

Boucle DOWHILE :

La boucle DOWHILE est aussi presque similaire à la condition IF dans le sens où il y a un tableau indexé de boucle while, il faut tester la condition, si elle est fausse alors il y a un saut à l'étiquette de fin de boucle. Sinon, il y a exécution du code du bloc d'instruction et un retour à l'étiquette du début de la boucle.

Boucle DOFORI :

La boucle DOFORI est la boucle pour avec incrémentation par une expression donnée par le programmeur.

Le symbole servant comme compteur d'incrémentation de la boucle ne doit pas avoir déjà été ajouté à la table des symboles. Ensuite, à ce symbole est affectée une expression qui permet de l'initialiser. Ainsi, cette variable est locale à la boucle, avec une valeur entière et son numéro de portée. À cet endroit, il y a une étiquette indiquant le début de la boucle. Je l'ai placé ici car juste après, il y a l'expression qui permet de tester si la condition de la boucle est vraie pour exécuter le code de la boucle. Par conséquent, si la condition est vraie, il y a donc exécution du code et ensuite, il y a un saut pour revenir à l'étiquette du début de la boucle. Sinon, c'est un saut à l'étiquette de fin de la boucle.

De même que l'instruction IF, IF ELSE et DOWHILE, la boucle DOFORI a un tableau permettant d'indexer la boucle pour les boucles imbriquées qui indique à l'étiquette son numéro unique correspondant à la boucle.

Affectation d'une variable :

L'affectation d'une variable n'est pas une déclaration comme en C, j'ai choisis de l'indiquer comme une instruction d'assignement, car la variable peut être redéfinie quand le programmeur le souhaite, mais aussi car il y n'y a pas de typage de variable.

Lorsqu'il y a une affectation, il y a une recherche dans la table de symboles si le symbole à affecter doit avoir une nouvelle valeur, car il existe déjà ou bien être ajouté à la table avec sa valeur. En code assembleur, il y a dépilement de la valeur de la pile et affectation de cette valeur à la variable.

Incrémentation d'une variable :

L'incrémentation d'une variable se réalise que si le symbole représentant la variable existe dans la table des symboles et ainsi il y a addition de un à la variable en code assembleur.

Décrémentation d'une variable :

La décrémentation d'une variable se fait que si le symbole représentant la variable existe dans la table des symboles et ainsi il y a soustraction de un à la variable en code assembleur.

Expression :

Pour chaque expression, le résultat de l'expression est empilé sur la pile. C'est ainsi que les instructions dépilent et utilisent la valeur d'une ou des expressions qui leur sont nécessaires.

L'expression parenthèse ne fait rien si ce n'est qu' elle remonte juste le résultat de la règle dans la référence du résultat de l'expression.

Pour les expressions binaires, il y a une comparaison des types. Dans notre sujet, le type est toujours entier, mais par bonne pratique, je teste le type et si le type représentant l'expression 1 ou 2 ne sont pas équivalents alors il y a remonté du type ERROR_T dans la référence du résultat de l'expression.

L'expression + (addition) dépile les deux valeurs de l'expression puis fait l'addition et met le résultat sur la pile.

L'expression - (soustraction) dépile les deux valeurs de l'expression puis fait la soustraction et met le résultat sur la pile.

L'expression * (multiplication) dépile les deux valeurs de l'expression puis fait la multiplication et met le résultat sur la pile.

L'expression / (division) dépile les deux valeurs de l'expression puis fait la division et met le résultat sur la pile. Une erreur est générée avec un message s'il y a une division par zéro. La variable assembleur indiquant l'erreur de division par zéro est à chaque fois conçue même s'il n'y a pas d'expression de division rencontrée. J'aurais dû mettre la variable avec un saut de variable (pour les boucles pour pas recréer la variable et donc faire une erreur) et la créé lorsque l'expression de division est rencontrée mais j'ai décidé de le laisser dans le programme run.

L’expression % (modulo) dépile les deux valeurs de l’expression puis fait le modulo et met le résultat sur la pile. Le modulo est fait en utilisant la formule :

(nombre - diviseur * (nombre / diviseur))

L’expression EQ (égale) dépile les deux valeurs de l’expression puis compare si les deux valeurs sont équivalentes et met le résultat sur la pile.

L’expression NEQ (différent) dépile les deux valeurs de l’expression puis compare si les deux valeurs ne sont pas équivalentes et met le résultat sur la pile.

L’expression AND (et) dépile les deux valeurs de l’expression puis utilise l’opérateur logique ET et met le résultat sur la pile.

L’expression OR (ou) dépile les deux valeurs de l’expression puis utilise l’opérateur logique OU et met le résultat sur la pile.

L’expression ! (négation) dépile la valeur puis la compare avec zéro. Si la valeur est égale à zéro, c’est 1 (vrai) qui est empilé, sinon c’est 0 (faux).

L’expression > (plus grand que) dépile les deux valeurs de l’expression puis compare si la seconde expression est plus petite que la première. Si c’est vrai alors c’est 1 qui empilé, sinon c’est zéro et met le résultat sur la pile.

L’expression < (plus petit que) dépile les deux valeurs de l’expression puis compare si la première expression est plus petite que la seconde. Si c’est vrai alors c’est 1 qui empilé, sinon c’est zéro et met le résultat sur la pile.

L’expression NUMBER (nombre/chiffre) met le chiffre ou le nombre rencontré sur la pile.

L’expression IDENTIFIER (symbole représentant une variable) recherche dans la table des symboles s’il est inexistant dans la table des symboles. Si c’est faux, alors il y a chargement de la valeur de la variable et met le résultat sur la pile.

L’expression IDENTIFIER argument_expression_list sert pour l’appel de fonction. Il y a vérification que le symbole (IDENTIFIER) est une fonction et qu’elle existe. Ensuite, une nouvelle vérification s’impose concernant le nombre d'arguments de la fonction qui doit être exactement celui attendu par la fonction. C’est ensuite qu'il y a un appel à la fonction et après un dépilement de ses arguments. Le résultat de la fonction est par convention stocké dans le registre ax. Par conséquent, le résultat du registre ax est mis sur la pile.

À propos de la commande \SIPRO, ce programme récupère en argument le nom de la fonction avec ses arguments par l’utilisateur s' il y en a. Comme le programme algo2asm a écrit le code assembleur correspondant à la description de l’algorithme, le programme run récupère ce code pour l'additionner à son code contenant le code principale assembleur avec l’appel de la fonction et ses arguments données par l’utilisateur. C’est alors, que le fichier généré par le programme run porte le nom de la fonction avec le supplément main indiquant que c’est le programme principal. Ainsi l’utilisateur utilise asipro et sipro pour lancer le programme principal se suffixant par main.asm.

3. Difficultés rencontrées

Durant le développement de ce projet, j'ai rencontré plusieurs difficultés et dont quelques-unes persistent encore.

Tout d'abord, concernant l'algorithme récursif, ce problème persiste car j'ai créé un test avec un algorithme factorielle qui de 1 à 7 fonctionne mais au delà le résultat est erroné et je n'ai pas réussi à trouver la solution. De plus, l'algorithme de Fibonacci ne fonctionne pas et je n'ai pas réussi à trouver la solution.

Après, j'ai eu le problème de l'élaboration de la grammaire tout entière. Au début, la grammaire n'était pas claire. J'ai retravaillé pas à pas pour que chaque fonctionnalité fonctionne. Pour cela, j'ai dû réparé les règles erronées grâce à la fonction printf pour savoir si c'était reconnu dans l'analyse lexical (Flex) et grammaticale (Bison).

Un autre problème que j'ai rencontré était la libération des éléments de la table des symboles. Je me suis dis que je devais libérer les symboles tout à la fin du programme. Or, cela est incorrect car cela voudrait dire qu'une variable pourrait être utilisée à n'importe quel moment dans la fonction. J'ai donc regardé mon cours de compilation, pour reprendre où se faisait la libération des symboles lorsque la fonction est terminée ou bien que le bloc est achevé.

4. Conclusion

Pour conclure, ce projet m'a permis de compléter mes connaissances pratiques et théoriques acquises en travaux pratiques, travaux dirigés et cours magistraux. De plus, j'ai acquis de nouvelles compétences techniques que je n'avais pas terminé d'apprendre en travaux pratiques comme l'implémentation de la boucle for et while grâce aux ressources indiquées par mes professeurs.

En termes de fonctionnalités, il reste beaucoup à faire pour que cela soit un traducteur accompli en termes d'analyse grammaticale, lexicale et d'optimisation du code assembleur (code à 3 adresses le support du cours).