TAURAND Sébastien
BERTRAND Antoine

PROJET INFORMATIQUE 2ème année

ASSEMBLEUR MIPS 32 BITS

Livrable 2 : Analyse syntaxique

Introduction

La seconde étape du projet consistant à réaliser un assembleur MIPS est de réaliser l'analyse syntaxique du fichier d'entrée. L'objectif est de parcourir la liste de lexèmes obtenue précédemment et de vérifier la validité de la syntaxe des phrases de lexèmes.

Organisation du travail

Sur cette phase du projet nous nous sommes réparti le travail de la façon suivante : l'un s'est occupée de faire les différents traitements liées aux sections .data et .bss tandis que l'autre s'est occupé des traitements liées à la section .txt et à la table d'étiquettes. Nous avons finalement mis en commun notre travail afin de réaliser l'automate final de l'analyse syntaxique.

Les Outils

La documentation du code et des structures de données a été réalisée à l'aide de Doxygen.

Nous avons choisi d'utiliser l'environnement de développement intégré Eclipse. Il nous a permis une meilleure ergonomie, rapidité de vérification, compilation tems réel, suggestion des champs objets déjà définis, intégration de git. Après une période d'apprentissage, il s'est avéré très utile et efficace et propose un débuggeur performant et facile à utiliser. Valgrind a également été utilisé de façon intensive au vu des nombreux pointeurs à gérer.

L'analyse syntaxique

L'analyse syntaxique du fichier assembleur est effectuée sur la base de la liste des lexèmes générée par l'automate lexical

Afin de simplifier le traitement des valeurs numériques, nous avons du légèrement retoucher à la machine d'état lexicale au niveau du traitement des signes '+' et '-'. Ils étaient auparavant des lexèmes séparés précédent les lexèmes nombres associés. Nous avons utilisé la fonction de conversion numérique à partir d'une chaine `strtol` qui a comme énorme avantage par rapport à sscanf de gérer proprement les erreurs. Elle permet aussi de travailler en base automatique (décimal, octal, …) ce qui nous aura permis en sortie du traitement lexical d'unifier la catégorie de l'ensemble des lexèmes numériques.

Nous avons aussi fait un comptage des étiquettes contenues dans le code dans l'analyse lexicale ce qui permet de ne pas avoir à redimensionner la table devant les contenir.

La structure de stockage des listes de lexèmes a été modifiée en une unique structure de liste générique afin d'en faciliter la manipulation. Le module liste.h sert aussi à stocker et gérer les données des sections .data et .bss ainsi que les instructions de la section .text.

Pour les collections de données ne se prêtant pas de façon efficace à un parcours systématique afin de pouvoir y effectuer des recherches, nous avons mis en balance deux alternatives :

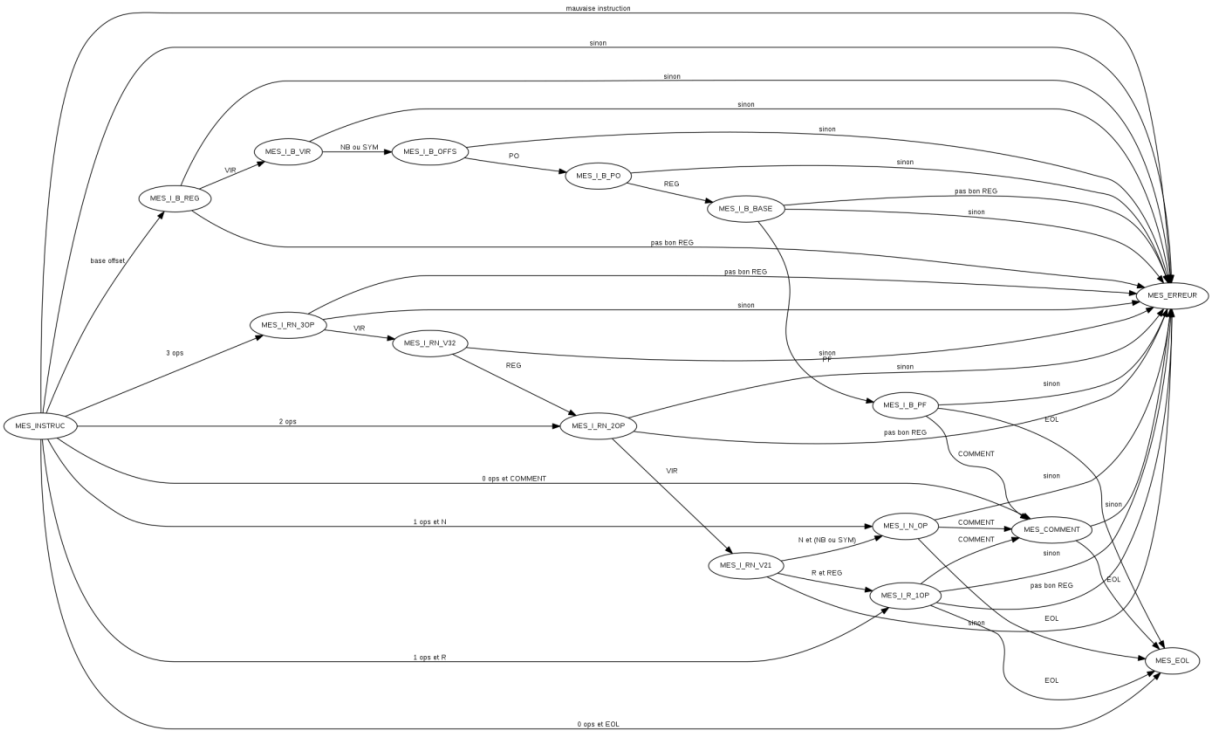
- Une collection sous forme de tableau trié et parcouru de manière dichotomique (complexité log(n))
- Une collection sous forme de table de hachage plus complexe, en particulier dans sa gestion des collisions, mais plus performante et plus souple à l'utilisation.

Nous avions commencé à coder le dictionnaire des instructions suivant cette première méthode, et les instructions pouvant elle-même être rangées par ordre alphabétique dans le fichier dictionnaire. Toutefois suite au besoin de stockage et d'accès efficace aux étiquettes

nous avons finalement implémenté une méthode de table de hachage avec gestion des collisions basée sur un double hachage et une taille de table première. Le dictionnaire d'instructions a aussi été porté pour utiliser cette méthode générique grâce au module table.h.

L'analyse syntaxique est basée sur un automate à état finis traitant l'ensemble des cas et états. La chaine de lexèmes est parcourue de son début à la fin, les chaines principales de l'automate étant calquées sur la grammaire du langage assembleur essentiellement limité à une seule ligne. Les instructions qui sont en fait le premier symbole en début de ligne sont convertis en majuscule afin d'être plus tolérant aux erreurs ou aux imprécisions de casse. Les directives, qui commencent par un '.' sont converties en minuscules, tout comme les noms de registres commençant par un '$'. Seuls les symboles ne sont pas modifiés et restent donc « sensible à la casse ».

La partie la plus complexe de l'automate correspond à l'analyse d'une ligne d'instruction :



Machine à états finis d'analyse syntaxique

Ce diagramme d'états, au-delà de son esthétisme permet de visualiser le volume et la complexité de la gestion des erreurs (les conditions de transitions en erreur ou « sinon »). Nous avons essayé d'être le plus simple et le plus explicite possible dans les messages d'erreurs générés en cas d'erreurs de syntaxe afin de faciliter la lecture de l'utilisateur, cela reste toutefois perfectible.

Nous avons implémenté l'auto-alignement des étiquettes en vérifiant si elles précédent immédiatement (à part autres étiquettes, commentaires et fin de ligne) un directive .word. C'est une méthode simple et robuste, tant qu'il n'y a pas de changement de section entre la déclaration d'une étiquette et le .word suivant (ce qui ne serait pas une très bonne pratique de programmation). L'automate est tolérant à plusieurs étiquettes se suivant, même sur la même ligne, y compris sur une ligne contenant une instruction ou directive (on repasse à l'état INIT après chaque étiquette rencontrée).

Nous faisons aussi une vérification de validité de nom de registre, sur la base d'un dictionnaire de type table de hachage générique.

Les données tolèrent un nombre non restreint d'opérande après la directive tant qu'il y a les virgules de séparation. Le stockage de ces données multiples est effectué sous forme d'une liste d'éléments uniques de donnée. Le stockage utilise un identificateur de type de donnée ainsi qu'une structure union pour pouvoir y accéder.

Seul les données asciiz ne sont pas encore traiter suite à une omission d'une la machine lexicale, cette omission sera corrigée pour le livrable 3.

Test réalisés

Nous avons essentiellement travaillé avec deux fichiers :

Un contenu de façon exhaustive l'ensemble des instructions et autres éléments de syntaxe correct. L'autre explorant de façon systématique des erreurs sur nom d'instruction, type des paramètres, valeur des paramètres, …

Nous avons vérifié avec valgrind que l'ensemble de la mémoire allouée était restituée précisément en fin de programme. Les seuls cas où la mémoire pourrait ne pas être restituée seraient sur des erreurs d'allocation mémoire dus à une mémoire saturée, chose pour laquelle nous nous contentons de sortir en ERROR_MSG.

Problèmes rencontrés

Le fichier tutotat2.pdf ne nous a malheureusement pas été présenté en cours. Ce document que nous avons découvert deux semaines plus tard nous a aidé de façon significative, mais nous avions déjà accumulé un retard important, dû à une méthode « semi-empirique » ressemblant à une machine d'état mais sans en avoir le formalisme et la rigueur. Nous nous sommes rendus compte de notre erreur au cours de l'enrichissement de ce code qui devenait ingérable, car très difficile à analyser. Nous avons donc pris la décision COURAGEUSE de jeter une grande partie du travail antérieur afin de repartir sur une base plus rigoureuse et saine.

Le lexème CHAINE_DE_CARACTERE n'étant pas traité dans l'analyse lexicale par manque de temps les directives .asciiz n'ont pas pu être traités.

Le projet demande un investissement de temps et d'énergie considérable.

Résultats

Le programme de test documenté, doté de génération de traces principales est testé avec un échantillon de fichiers tests nous semble essentiellement fonctionnel et répondre aux attentes, voire les dépasser sur certains points (vérification de la validité des registres et des symboles non définis).

Points à améliorer

Séparation de l'automate principal en sous-automates de taille plus raisonnable.
Implémentation du traitement des directives .asciiz.

Documentation du fichier syn.c.