Gwendal Jouneaux : chef de projet - responsable documentation - expert NASM
Timothée Schneider-Maunoury : responsable tests - expert code intermédiaire

Rapport de fin de projet
Compilateur de While en NASM

ESIR 2 SI - promotion 2017/2020

Introduction :

Durant le septième semestre de notre parcours à l'ESIR, nous avons eu l'opportunité de choisir un langage comme langage cible de notre projet de compilation. Notre choix s'est porté spontanément vers un langage assembleur, famille de langages de très bas niveau connus pour la complexité de leur utilisation par les programmeurs. C'est bien sûr l'attrait du risque et du défi qui a motivé notre décision, mais aussi la volonté de comprendre bien plus en profondeur certains mécanismes des systèmes informatiques à travers ce langage proche de la machine.

Ce document va donc décrire les rouages de notre compilateur du WHILE : langage très simple mais très complet, au Netwide Assembler ou NASM : langage avec un large ensemble d'instructions et pouvant produire des programmes portables sur linux et windows. Nous détaillerons également l'organisation de notre équipe et la manière dont notre projet à évolué au long de ce semestre.

Avant d'entrer dans le vif du sujet, nous tenons à remercier Olivier Ridoux et Morgan Gautier qui nous ont suivis et aidés pendant tout le projet.

Table des matières :

I. Description technique

I.A. Schéma d'exécution

I.A.a. Prélude

Le prélude du programme est une partie très complexe. En effet, c'est lui qui va lire les paramètre passé par l'environnement shell au programme, les convertir en arbres binaire et appeler la fonction main. Dans notre projet, il sert aussi de mise en place des différentes données nécessaires au fonctionnement du programme. Cette partie à été un point relativement difficile, car les paramètre passés à un programme en assembleur se retrouve dans la pile, il faut donc les retrouver pour pouvoir ensuite les utiliser. Pour cela, il faut commencer par regarder le nombre d'arguments, par chance cette information se trouve sur le dessus de la pile. Ensuite, il faut suivre l'adresse rangée après le nombre d'arguments pour obtenir les arguments séparé par un caractère nul.

I.A.b. Appels de fonction

Une fois le prélude exécuté, la fonction "main" doit s'exécuter. Nous avons choisis de créer une méthode de passage d'argument générique permettant au prélude de passer des paramètres à la fonction "main" exactement comme le ferait la fonction "main" pour appeler d'autres fonction. Après réflexion, la solution qui pour nous était la plus adaptée était le passage de paramètres par une file. Nous avons aussi remarqué que la file était toujours en écriture puis en lecture, mais qu'il n'y avait pas de données ajouté tant que la lecture des dernières données n'avais pas été faite. Nous avons donc créé un buffer et deux pointeurs, un pour l'écriture et l'autre pour la lecture. les fonctions read et write permettent donc de lire et écrire des données dans le buffer et gèrent les pointeurs, notamment la remise à 0 de pointeur d'écriture lors de la lecture et vice-versa.

I.A.c. Affectations

Dans la première partie du projet, les affectations était des affectation simples. La solution la plus logique était donc de créer un code 3@ "assign" permettant de réaliser l'affectation simple. Mais par la suite, nous avons ajouté la notion d'affectation multiple et simultanées. La solution qui se proposa à nous, fut d'utiliser le buffer de fonction puisqu'en réalité une affectation multiple est

comme une fonction prenant N paramètres et ne faisant rien hormis retourner ses paramètres.

A,B,C = E,F,G ⇔ A,B,C = (returnParam E F G)

Nous avons donc traduit l'affectation comme une suite d'écriture dans le buffer suivi d'une lecture du buffer, De cette façon nous avons réussi à homogénéiser l'affectation avec l'appel de fonction ce qui à rendu trivial le problème d'affecter les multiple valeur de retour d'une fonction. Cela s'est révélé d'autant plus pratique par la suite, car l'affectation et l'appel de fonction dupliquent des références, l'utilisation d'une même fonction (write) pour les deux cas à donc rendu le travail de comptage de référence beaucoup facile à gérer.

Il est à noter que notre compilateur est très complet dans la détection des erreurs de types (plus précisément : des erreurs liées à l'arité des opérations) au sein des affectations et/ou des appels de fonctions. L'ensemble des erreurs possibles provoque un arrêt de la compilation avec un message détaillant la cause de l'erreur, ainsi aucune erreur ne peut se produire à l'exécution !

I.A.d. Expressions

Les expressions hd et tl font simplement appel à une procédure d'assembleur qui va les évaluer en modifiant la composition des noeuds dans la mémoire, puis les enregistrer.
Pour les expressions cons et list à plus de deux paramètres, un découpage est effectué lors de la production du code intermédiaire afin de tout ramener à une suite de cons à deux paramètres, ce qui va être traité comme hd et tl lors de la génération de code assembleur, c'est à dire en déclenchant une procédure.

Une particularité intéressante de notre compilateur se situe dans sa mise en oeuvre efficace du système de court-circuits qui entremêle les expressions booléennes et les structures de contrôles if et while.
Cette méthode consiste à transmettre des étiquettes siVrai / siFaux aux expressions booléennes qui vont ainsi pouvoir aller directement à l'endroit du flot d'instructions qui corresponds à la valeur résultant de leur évaluation.
Par exemple, une commande if va transmettre les étiquettes de son "then" et de son "else" à l'expressions conditionnelle. Celle-ci va s'évaluer et fair un "jmp" qui va déplacer le compteur ordinal des instructions à "then" ou "else" en fonction de son résultat. Ce mécanisme permet de ne pas avoir à évaluer l'expression et stocker son résultat dans une variable d'un côté, puis consulter la variable et se déplacer en conséquence d'un autre côté.
L'évaluation des expressions or et and est également optimisée avec des courts-circuits en faisant en sorte que le minimum d'expressions contenues dans

le or/and soit évalué. Si on trouve une expression vraie (resp. fausse) dans une expression or (resp. and), on va directement à l'étiquette siVrai (resp. siFaux) au lieu de continuer inutilement à évaluer les autres expressions.

I.A.e. Structures de contrôle

Toutes les structures du langage While sont analysées au moment de produire du code intermédiaire, et retranscrites en une suite d'étiquettes et de goto.

Comme nous l'avons précisé précédemment, les commandes if et while utilisent simplement le système de court circuit en transmettant leur étiquettes. Les siVrai et siFaux correspondent aux "then" et "else" du if, et au "boucle" et "finBoucle" du while.

Les commandes for et foreach sont un peu plus complexes : afin de s'assurer que l'expression qui détermine le nombre d'itérations de la boucle puisse être utilisée dans le corps de la commande sans en perturber l'exécution, on commence par créer une copie de cette expression.

Ensuite on compare cette copie à 0, puis si on la décrémente à chaque itération avec la procédure tail jusqu'à ce qu'elle soit nulle, ce qui provoque la sortie de la boucle. Dans la commande foreach, on ajoute la procédure head avant de décrémenter pour extraire successivement toutes les valeurs de l'expression correspondante.

I.A.f. Postlude

Le postlude est pour dire les choses simplement l'inverse du prélude. Le postlude se charge de convertir les arbres binaire en entier ou chaîne de caractères et les renvoie à l'environnement d'appel, c'est à dire le shell. Ici, cette conversion est simplifiée car elle précisé dans le HEAD de l'arbre retourné. Il n'y a donc pas à déterminer le type de retour car il est déjà présent. Notre postlude, se contente donc de lire un à un les arbres de sortie, contrôler le type en comparant son HEAD avec les symboles "int" ou "string", puis de convertir et d'afficher ces arbres dans la console.

I.B. Architecture logicielle

I.B.a. Grammaire de WHILE

La première brique de notre projet fût la grammaire de WHILE. Durant notre cours de Théorie des Langages, nous avons réalisé une grammaire de WHILE en utilisant l'outils xText. Ces grammaires ont été le point de départ de la construction de notre grammaire finale plus complète et nous permettant de

pouvoir récupérer correctement les informations nécessaires à la compilation d'un fichier WHILE.

I.B.b. Xtext

Xtext est un outil très complet qui permet de faire beaucoup de choses dans le domaine des langages dédiés (ou Domain specific language en anglais). Nous avons utilisé pour ce projet quelques unes de ses nombreuses fonctionnalités que nous allons vous décrire dans cette partie.
Xtext commence par parcourir le programme while à compiler pour construire son Arbre de Syntaxe Abstraite correspondant en fonction de la grammaire que nous avons spécifiée.
Nous allons ensuite parcourir chaque fonction de cet arbre et lui associer un objet de type DefFunction contenant : un nom normalisé, son nombre de paramètres et de retours, une table des variables et surtout la traduction de cette fonction en une liste d'instructions en code intermédiaire.
La fonction main de notre compilateur contient une table des fonctions qui rassemble toutes ces instances de DefFunction, ainsi qu'une table des symboles (les symboles ayant une portée s'étendant à tout le programme).
Une fois que la sémantique du programme while à compiler se retrouve dans ces structures de données, dispersée en plusieurs listes de code intermédiaire, on applique une fonction d'amélioration sur chacune de ces listes.
La fonction d'amélioration que nous avons implémenté analyse les affectations et utilisations des variables du code intermédiaire dans le but de détecter et éliminer les portions de code mort. Cette fonction est relativement complexe et efficace, notamment, elle prends en compte la présence de boucles dans les instructions. Ainsi une évaluation d'expression et son affectation à une variable sont supprimées seulement si la variable en question n'est pas susceptible d'être utilisée plus tôt dans le flot d'instruction lors de l'exécution.
Finalement, ce code intermédiaire amélioré est traduit instruction par instruction en code NASM et inséré entre le prélude et le postlude, sans oublier l'importante bibliothèque permettant de faire fonctionner le code assembleur généré.

I.B.c. Librairie BinTree

Pour compiler du WHILE en assembleur, il nous a fallu construire une librairie pour gérer la structure d'arbre binaire, qui est le seul et unique type en WHILE. Cette librairie à donc entièrement été écrite en assembleur et contient l'ensemble des fonction étant liée aux arbres binaire on retrouve notamment les fonction **cons**, **head**, **tail** et **equals**. On retrouve également dans cette bibliothèque les fonction liée au fonctionnement de WHILE comme par exemple **read**, **write** et les fonction de gestion de la mémoire.

I.B.d. Librairie Launcher

En plus de la librairie BinTree, nous avons fait le choix de créer une librairie en rapport avec le lancement du programme. En effet, le lancement étant fastidieux en assembleur cette librairie recense les fonction utile au prélude et au postlude tel que les fonction de conversion entre les arbre binaires et d'autres type. Ces fonctions permettent de convertir les paramètres en arbres et inversement à convertir les arbres binaires en informations utiles ou plus lisibles. Cette librairie contient aussi l'affichage d'arbre binaire et la fonction de parsing d'une expression de type "(cons (cons a b) c d e)", la librairie possède donc évidemment aussi une fonction permettant d'ajouter de nouveaux symboles au programme.

I.B.e Tests automatiques

Afin de détecter des bugs dans notre compilateur et plus particulièrement de s'assurer de la non-régression de notre projet, nous avons implémenté toute une série de tests pilotés par un script bash de plus de 600 lignes !
La majorité des tests consistent à compiler un programme while puis l'exécuter et comparer son résultat à un oracle pour déterminer si le test est validé ou non.
Nous aussi testé si le compilateur détecte bien toutes les erreurs de types que l'on peut retrouver dans un programme while.

II. Bilan de gestion de projet

II.A. Étapes du développement

Lors de ce projet, nous avons suivi une méthodologie agile. Ce projet à donc été découpé en 3 sprints, et chaque sprint était composé de plusieurs reviews permettant de présenter au Product Owner l'avancement du projet et pouvoir ainsi obtenir des retours sur notre travail.

Durant le premier sprint, nous avons tout d'abord évalué les risques potentiels du projet (la taille réduit de l'équipe et le langage cible assembleur) et défini un dossier de spécification reprenant l'ensemble des fonctionnalités attendues pour le sprint regroupées en versions. Nous avons ensuite mis en place des outils pour collaborer tel qu'un tableau de suivi (Trello) et un système de versionnage du projet. Nous avons également pensé à utiliser des outils d'intégration continue, mais nous avons jugé que le jeu n'en valait pas la chandelle au vu de la taille du projet et avons donc mis en place nos propres outils, comme par exemple un script de test automatique écrit en bash.

Au cours de notre première rétrospective de fin de sprint, nous avons constaté que la répartition des rôles n'était pas optimale. Nous avons donc commencé le second sprint avec une répartition plus homogène des tâches. Nous avons décidé lors de ce second sprint de commencer avec un processus légèrement moins incrémental , car la réalisation d'une première librairie BinTree ainsi que la construction du lanceur en assembleur allait prendre un temps conséquent, malgré le fait que cela soit deux éléments indispensable au lancement d'un programme de WHILE compilé en assembleur. Une fois ce travail terminé et l'intégration de ces élément avec les première instructions compilées, nous sommes repartis sur une méthodologie agile. Nous avons par la suite pu apprendre que cette façon de faire est appelé "Scrum but but". C'est à dire que l'on fait de l'agilité sans certaine chose ("Scrum but") mais pour une raison précise, dans notre cas la nécessité d'avoir une librairie et un lanceur complet pour pouvoir compiler un programme WHILE aussi simple soit-il.

Le second sprint se passant bien pour nous, nous avons pu anticiper les problèmes du troisième sprint et consolider notre méthode. C'est pourquoi lors de la rétrospective du second sprint la seule remarque que nous avions trouvé était le manque de tests, ce qui à été corrigé dans le sprint suivant en commençant le développement des test dès le début.

Notre méthode ayant été validé au second sprint, le troisième sprint s'est déroulé sans problèmes particulier, nous avons même pu ajouter des amélioration sur la fin du sprint tel que le comptage de référence ou la suppression de code mort.

II.B. Rapports d'activité individuels

II.B.a. Gwendal

II.B.a.1. Gwendal chef de projet

En tant que Chef de projet, ma première mission a été de répartir les tâches et d'organiser l'équipe. Cette mission n'as bien sûr pas été compliqué puisque nous n'étions que deux dans l'équipe, c'est donc sur une entente commune que s'est fait le découpage des tâches et d'un commun accord que nous avons choisis les outils utiles à notre projet. Durant l'ensemble du projet, je m'assurais que nous respections le plus possible notre méthodologie en organisant les rétrospective. Dans l'ensemble mon travail de chef de projet ne fut pas trop dur étant donné le nombre de collaborateurs réduit, la communication ne posait

aucun problème et nous étions donc constamment au courant de l'état global du projet.

II.B.a.2. Gwendal responsable de la documentation

En tant que responsable de la documentation, je m'assurais que les spécification ainsi que l'ensemble des documents techniques était à jours et le cas échéant les mettais à jour dans la limite de mes connaissance sur le point technique en question. Tout comme dans mon rôle de chef de projet, ce travail fut relativement simple, car Timothée et moi écrivions notre documentation de manière régulière et claire.

II.B.a.3. Gwendal expert assembleur

Mon plus gros travail dans ce projet était d'être l'expert en assembleur du groupe. C'est à dire que j'ai dû écrire l'ensemble des deux librairies, les débugger et parfois les revoir complètement à cause du changement d'un choix d'implémentation. A la fin du projet, l'ensemble du code assembleur produit atteint les 1137 lignes et cela sans compter les nombreuses ligne qui n'auront pas survécu au projet.

Ma première tâche fut de choisir quel langage d'assemblage nous allions choisir. Plusieurs option avait été envisagé notamment l'assembleur MIPS qui nous était inconnu ou l'assembleur du processeur du cours d'architecture (ARC) qui était trop peu complet. Nous avons donc opté pour le NASM, un langage d'assemblage relativement récent utilisant une syntaxe Intel que nous connaissions grâce à notre cours de Système de l'année passé, en plus, nous pouvons assembler NASM pour nos machines directement.

Ma seconde tâche fut donc de concevoir la librairie BinTree, j'ai commencé par celle-ci car son implémentation et la structure des arbres binaire allait dicter la façon dont construire le launcher par la suite. J'ai donc commencé par définir les structures de données tels que les noeuds d'arbres binaires ou les descripteurs de symboles, j'ai ensuite codé les fonctions liées à ces structures.

J'ai ensuite réalisé le début de la librairie Launcher en réalisant le parsing des arguments et en me contentant de traduire les nombre en arbres binaires et inversement. A ce stade j'ai du réfléchir à la connexion entre le launcher et la fonction "main". Comme expliqué plus haut, j'ai souhaité que cette connexion soit identique à celle de n'importe quelle fonction. Après réflexion, je suis arrivé avec l'idée du buffer que nous avons utilisé par la suite et qui me semble encore aujourd'hui la meilleur idée que j'ai eu sur ce projet.

Pour finir, j'ai amélioré l'ensemble de ces librairies et ai même ajouté quelques améliorations tel que le comptage de références.

II.B.b. Timothée

II.B.a. Timothée responsable des tests

Le rôle de responsable des tests à été très intéressant pour plusieurs raisons. Premièrement il m'a permis de renouer avec le scripting en bash que je n'avais pas pratiqué depuis relativement longtemps. Gwendal et moi développant sous Linux, le choix du script shell s'est fait naturellement et s'est avéré efficace : j'ai pu implémenter de nombreux tests automatiques rapidement et ainsi couvrir l'ensemble des instructions du langage while.

Les différents tests m'ont également amenés à écrire des programmes while de plus en plus complexes, et j'ai été ainsi étonné de découvrir l'étendue de ce que l'on pouvait faire avec un langage aussi réduit. J'ai finalement développé toute une librairie d'opérations arithmétiques allant de l'addition au pgcd et de la soustraction à la racine carrée. Le travail sur cette bibliothèque m'a d'ailleurs permis de trouver des cas de tests auxquels je n'avais pas pensé en premier lieu, et qui ont effectivement soulevé quelques bugs dans notre compilateur.

Plus largement, concevoir les tests m'a fait réfléchir plus concrètement aux fonctionnalités du compilateur qu'il nous restait à développer et a donc contribué à donner une ligne directrice à ce projet.

II.B.b. Timothée expert en code intermédiaire

La plus large part de mon travail a été d'imaginer les opérations du code intermédiaire puis de traduire le code while en une suite de ces instructions. J'ai choisi un jeu d'instructions simple de seulement 14 opérations afin de faciliter par la suite la traduction en NASM.
Durant cette exercice de traduction, J'ai été particulièrement intéressé par l'implémentation des courts-circuits qui a nécessité une réflexion logique assez importante, d'abord pour les expressions or et and, puis pour les instructions de contrôles.
J'ai également dû résoudre le problème de la vérification de type qui était assez simple dans le cas d'une affectation multiple, mais devenait plus complexe quand c'est une fonction qui était à droite de l'affectation.

Finalement, la mise en oeuvre de l'amélioration de suppression de code mort était captivante dans la mesure où il fallait vraiment imaginer une solution de toute pièce pour appliquer la théorie que l'on avait vu en cours.

Conclusion :

Ce projet fut globalement une réussite. Du point de vue de la gestion de projet, nous avons appliqué la plupart des préconisations de la méthode scrum ce qui a porté ses fruits car le compilateur a été fini à temps et avec toutes les fonctionnalités demandées.
Et du point de vue de la dynamique d'équipe, la communication et la collaboration a été efficace entre les deux membres du groupe, ce qui a permis d'avancer vite tout en se comprenant bien sur nos différentes tâches.
Finalement, travailler sur notre compilateur fut un réel plaisir et nous avons énormément gagné en compétences sur tous les aspects du projet.