IFT3065 - Proposition de projet

Vincent Foley-Bourgon (FOLV08078309)
Eric Thivierge (THIE09016601)

Janvier 2012

Résumé

Nous proposons d'écrire un compilateur Scheme en Scheme qui générera du code pour une machine x86. Ce compilateur sera une exploration "en largeur" du domaine de la compilation ; plutôt que d'implanter quelques fonctionnalités et de les développer en profondeur, nous allons tenter d'implanter plus de fonctionnalités, même si cela implique de diminuer le degré de sophistication de notre compilateur. Les deux membres de l'équipe pensent que cela aidera à obtenir une bonne vision d'ensemble du fonctionnement et des défis d'un compilateur.

1 Choix des langages

1.1 Langage source (Scheme)

Nous avons décidé d'utiliser Scheme comme langage source. La raison principale est que Scheme est un des langages de haut niveau les plus simple qui soit, ce qui permet d'implanter une grande partie du langage dans un court lapse de temps. De plus, il possède des fonctionnalités que d'autres langages, comme C ou Pascal, ne possèdent pas :

- Fermetures
- Symboles
- Récursivité terminale
- Continuations
- Macros

1.2 Langage hôte (Scheme)

Nous voulions un langage fonctionnel de haut niveau pour notre compilateur. Le langage OCaml était une option très intéressante,

étant donné qu'il est similaire au langage SML utilisé dans le livre "Modern Compiler Implementation in ML" par Appel. Son système de typage statique et sa facilité à manipuler des données symboliques auraient certainement étés forts utiles tout au long du développement. De plus, il possède des outils d'analyse lexicale (ocamllex) et syntaxique (ocamlyacc) si ceux-ci s'avèrent nécessaires. Comme un seul membre de l'équipe connaît OCaml, nous avons décidé de ne pas l'utiliser et avons donc choisit Scheme comme langage hôte.

Scheme est également un langage de fonctionnel de haut niveau qui permet des manipulations faciles de données symboliques. Écrire le compilateur dans le langage source crée un "cercle virtueux" d'apprentissage. De plus, nous avons la possibilité d'écrire un compilateur autogène.

1.3 Langage cible (assembleur x86)

Nous avons décidé de cibler le langage assembleur x86 pour notre compilateur. Malgré que les deux membres de l'équipe ne soient pas familiers avec cet assembleur, la popularité des machines x86 font de ce langage un choix pratique et la connaissance de l'assembleur x86 est un atout important. Le choix de x86 nous évite également de passer par un émulateur pour un autre assembleur (ex. : MIPS).

2 Choix des fonctionnalités

Afin de garder l'implantation du langage simple, nous avons éliminé quelques fonctionnalités de Scheme pour notre projet. Les fonctionnalités suivantes ne se trouveront pas dans notre langage :

- Les nombres à virgule flottante
- Les entiers à précision arbitraire
- Unicode
- `define-syntax`
- Les commentaires multi-lignes (`#| |#`)

Les fonctionnalités suivantes seront dans notre langage :

- *lambda* et les fermetures
- *cond*
- *let*
- Les paires
- Les listes
- La récursivité (optimisation des appels terminaux ?)
- Les commentaires commençant par un point-virgule
- Les entiers (fixnum)
- Les chaînes de caractères
- Les caractères

– Les symboles
– Les booléens
– Le mot clé *begin*
– Le mot clé *set !*
– L'instruction *do*

Nous ne sommes pas certains si nous pourront inclure les fonctionnalités suivantes :

– Les continuations
– Les macros
– Les nombres rationnels
– Les vecteurs

3 Design du compilateur

Nous avons décidé d'écrire un compilateur en pipeline très modulaire. On tentera de donner à chaque phase de la compilation son propre module et que ce module soit complètement indépendant. Cela affectera certainement la vitesse de compilation, mais nous sommes d'avis que la modularité du compilateur sera plus utile pour notre apprentissage et pour l'aboutissement du projet.

Cette approche est similaire à celle proposée par Sarkar, Waddell et Dybvig dans leur article *A Nanopass Framework for Compiler Education* [1]

Nous metterons peu d'efforts pour développer en grande profondeur une fonctionnalité particulière de notre compilateur, et nous préfèrerons explorer un plus grand nombre de fonctionnalités

A priori, notre compilateur ne sera pas autogène. Si le temps le permet, nous pourrons voir à la fin du projet s'il serait possible d'être auto-suffisant, mais ce n'est pas un objectif principal de notre projet.

4 Choix des tâches

4.1 Grammaire, parseur et messages d'erreurs (niv-2)

Comme nous implémentons notre propre compilateur, nous avons besoin d'un analyseur lexical et syntaxique. Nous avons décidé de les implanter manuellement plutôt que d'utililiser des outils comme lex ou bison, car nous aimerions connaître les détails et défis de plus bas niveau de ces phases. Par contre, nous réalisons que le gros morceau de la compilation se trouve dans le back-end plutôt que dans le front-end et que ce serait une erreur de dépenser trop d'efforts sur l'analyse

1. http ://www.cs.indiana.edu/ dyb/pubs/nano-jfp.pdf

lexicale et syntaxique. Nous procéderons donc de la façon suivante : nous tenterons d'implémenter notre propre lexer et parser à la main. Si une semaine après le début des travaux nous voyons que nous avons pris du retard et que nos lexer et parser ne sont pas fonctionnels, nous utiliserons des outils automatisés pour faire l'analyse pour nous.

Dans le cas où notre analyseur lexical est fait manuellement, il aura les propriétés suivantes :

- Support pour ASCII seulement : il serait trop long et difficile de supporter correctement l'ensemble de Unicode, et nous nous limiterons donc à l'ensemble ASCII.
- Le "type" de notre analyseur lexical sera `[char] -> [symbol]` ; on prendra en paramètre une liste de caractères et on retournera une liste de symboles. L'analyse lexicale ne sera pas concurrente avec l'analyse syntaxique.
- Un symbole sera un triplet de la forme : `(type+valeur, ligne, colonne)`. La ligne et la colonne seront utilisés pour donner des messages d'erreurs plus détaillés.

Notre analyseur syntaxique aura les propriétés suivantes :

- Ce sera un parseur LL(1) écrit par un algorithme de descente récursive, car c'est sans doute l'algorithme le plus simple à implanter manuellement.
- Le parseur prendra en paramètre la liste de symboles générée par l'analyseur lexical et retournera un arbre de syntaxe abstraite.

4.2 Fermetures (niv-2)

Les fermetures sont l'une des fonctionnalités les plus importantes du langage Scheme, et nous allons donc les implanter.

4.3 Récupérateur de mémoire automatique (niv-2)

Tout comme les fermetures, la gestion automatique de la mémoire est une fonctionnalité importante de Scheme. Nous allons donc implanter un récupérateur simple.

4.4 Types (niv-1)

Scheme est un langage typé dynamiquement, donc les variables ne possèdent pas de types, mais il est fortement typé, ce qui veut dire qu'une expression telle que `(+ 3 'foo)` est sémantiquement incorrecte. Nous traiterons ces erreurs de type.

4.5 Vivacité des variables (niv-1)

L'analyse de vivacité des variable permettra d'identifier les variables mortes (dont la valeur ne sera pas lue dans le future) et d'en retirer le

code dans le programme cible.

4.6 Propagation de constantes (niv-2)

On ajoutera une phase simple d'optimisation où les expressions constantes seront remplacées par leur valeur numérique.

Par exemple : transformer `(* 3 (+ 2 4))` par `18`.

4.7 Génération simple avec optimisation "peephole" (niv-2)

On écrira dans un fichier texte les lignes de code d'assembleur du programme source. On appliquera des optimisations simples sur le code assembleur. Par exemple, `mov $0, %ebx` sera transformé en `xor %ebx, %ebx`.