Rapport de l'étape 1 du projet compilateur
Étape 1

Francis de Ladurantaye
Thomas Luinaud

February 16, 2018

1 Spécification du problème

Le but du projet est de réaliser un compilateur pour un sous-ensemble sélectionné du langage de programmation Scheme. Le travail à réaliser étant divisé en trois étapes, nous devrons chaque fois étendre notre support à un plus grand sous-ensemble du langage, en construisant à partir du point où nous nous étions arrêté lors de l'étape précédente. À terme, le compilateur résultant devra être autogène, c'est-à-dire d'être en mesure de se compiler lui-même, de façon à garantir son autosuffisance en éliminant les dépendances aux compilateurs Scheme existants. Cela implique inévitablement que le sous-ensemble que nous chercherons à supporter au terme de la troisième étape devra être suffisamment étendu afin d'offrir la puissance nécessaire à l'atteinte de cet objectif mais également d'écrire le compilateur en prenant en compte les éléments que nous supporterons.

1.1 Sous-ensemble de Scheme à supporter

Au terme de l'étape 1 du projet, il est attendu de nous que le parseur produit soit habilité à supporter les constructions syntaxiques suivantes :

- Nombres entiers positifs et négatifs
- Booléens `#t` et `#f`
- Caractères, qui débutent par `#\`
- Chaînes de caractères et caractère d'échappement "\"
- Symboles
- Listes propres et impropres
- Commentaires

De son côté, le générateur de code devra permettre la compilation des expressions suivantes en se limitant pour l'instant simplement à la manipulation d'entiers et de booléens :

- Constantes entières
- Constantes booléennes
- `(println` $expr$`)`
- `(+` $expr_1$ $expr_2$`)`
- `(-` $expr_1$ $expr_2$`)`
- `(*` $expr_1$ $expr_2$`)`
- `(quotient` $expr_1$ $expr_2$`)`
- `(modulo` $expr_1$ $expr_2$`)`
- `(=` $expr_1$ $expr_2$`)`

- `(<` $expr_1$ $expr_2$`)`
- `(if` $expr_1$ $expr_2$ $expr_3$`)`
- `(let ((`var_1 $expr_1$`) ...)` $expr_0$`)`

1.2 Analyseur lexical (parseur)

En ce qui concerne la première étape du projet, il nous était demandé d'être en mesure de compiler et exécuter des fichiers Scheme simples ne comportant que les constructions syntaxiques du sous-ensemble de Scheme à supporter pour le moment. Cela implique que notre compilateur doit inclure un analyseur lexical (parseur) et syntaxique (générateur de code) utilisant l'approche par descente récursive, dont les rôles seront respectivement d'effectuer la lecture des fichiers à compiler, caractère par caractère, afin d'en extraire les différents éléments et de produire le code machine résultant. En plus d'extraire les éléments pertinents lors de se traversée (lecture), le parseur doit aussi effectuer le regroupement des éléments extraits en expressions (listing), simplifiant ainsi le travail du générateur de code assembleur qui n'aura donc nul besoin de retraverser ces fichiers à la suite du parseur.

Afin d'illustrer le travail du parseur, supposons que le contenu du fichier à compiler est le suivant : "Hello World!". Le parseur, lors de sa traversée dudit fichier, reconnaîtra le guillemet se trouvant au début et, sachant alors qu'il se trouve dans une chaîne de caractères, lira successivement les caractères suivants jusqu'à ce qu'il tombe sur le guillemet fermant à la suite duquel cet élément *chaîne de caractère* sera ajouté à une liste servant à représenter le sens du programme. Cette étape est très importante car, dans le cas présent, le caractère d'espacement au centre de la chaîne n'a ici pas d'autre sens que celui d'un simple caractère, mais il aurait signifié la délimitation entre deux éléments distincts s'il ne s'était pas trouvé entouré d'une paire de guillemets. De nombreux caractères ayant ainsi plusieurs sens, et ce dans tous les langages de programmation, un générateur de code devant effectuer cette tâche parallèlement à la production du code machine paraît quasi irréalisable, l'implantation d'un simple parseur représentant déjà un défi non négligeable.

1.3 Analyseur syntaxique (générateur de code)

Le générateur de code est l'élément central du compilateur. Une fois produit le *listing* des expressions, le générateur de code a tout ce dont il a besoin pour effectuer sa tâche. Cela s'explique par le fait que la représentation sous forme de *listing* des expressions du programme correspond au sens de celui-ci, de manière dissociée de la chaîne de caractères dont il provient. Le travaille du générateur de code se résume donc à parcourir le listing reçu et d'en traiter les éléments de la façon appropriée, ce pour en générer le code machine dont l'exécution produira les résultats correspondant au programme originel.

Cette tâche est cependant plus compliquée qu'il n'y paraît. En effet, le générateur se charge non seulement de la génération de code, mais aussi de s'assurer que les expressions reçues sont syntaxiquement correctes. Cela nécessite entre autres de vérifier que les procédures contenues

dans les expressions ont le nombre d'arguments attendu ou que les variables apparaissant dans les expressions ont bien été initialisées antérieurement à leur utilisation. En outre, c'est au générateur de code qu'il revient de gérer les fermetures lors de la présence de `let` et de s'assurer que l'accès au contenu d'une variable ayant ainsi été redéfinie renvoie la valeur attendue. Ces difficultés ne font que s'accumuler lorsque le programme à compiler contient de multiples `let` imbriqués, requérant une gestion de la pile appropriée à ces constructions syntaxiques qui sont tout à fait correctes.

Comme dernières complications liées au générateur de code assembleur, la gestion des symboles et des énoncés conditionnels. Au niveau de l'énoncé conditionnel `if`, la génération des étiquettes servant à sauter les blocs ne devant pas être exécutés doit être gérée de façon particulière car plusieurs `if` peuvent se trouver dans un même programme. Il est important que chacun d'eux n'effectue pas de sauts vers des portions de code des autres `if`. En lien direct avec les énoncés conditionnels, certains symboles sont particulièrement importants. Nous parlons bien entendu ici des symboles `#t` et `#f` qui doivent être reconnus par le générateur de code car, ne se trouvant pas dans l'environnement comme ceux associés aux variables ou référant à des procédures, une recherche dans l'environnement engendrerait la détection d'une variable non initialisée.

Toutes ces difficultés seront donc à prendre en considération tout au long de l'implémentation du générateur de code afin de s'éviter d'avoir de mauvaises surprises nécessitant de recoder le corps de plusieurs fonctions, et ce possiblement à de multiples occasions.

1.4 Tests unitaires

Le dernier élément du travail à réaliser était la conception de tests unitaires pertinents servant à vérifier le bon fonctionnement des différents éléments de notre compilateur, principalement du parseur et du générateur de code. En plus des 27 tests fournis, nous devions en produire entre 30 et 50 tests supplémentaires pour bien vérifier chaque fonctionnalité. Les résultats des tests seront présentés dans une section subséquente de ce rapport.

2 Méthodologie

En ce qui a trait à la méthodologie adoptée pour l'attente des objectifs, nous avons débuté par l'analyse des fichiers fournis afin de nous faire une certaine idée de la direction dans laquelle nous lancer. Nous avons testé de façon extensive la fonction `read` afin de bien comprendre ce qui était attendu comme résultat de la traversée d'un fichier par notre futur parseur. Ce faisant, nous avons aussi constaté les manques de cette fonction, nous donnant donc une idée assez claire sur ce qu'il restait à y implémenter.

Autrement, nous avons testé les différentes commandes offertes par le `makefile` fourni, ce qui nous permit rapidement de comprendre comment lancer le compilateur pour exécuter les tests unitaires. Nous avons donc rapidement pu écrire des tests supplémentaires, ce qui nous aida grandement au débogage lorsque nous tentions d'implanter de nouvelles fonctionnalités

et encore plus grandement lorsqu'il nous était nécessaire de réimplémenter une fonction après avoir constaté un problème.

Par ailleurs, nous nous sommes fréquemment référé à la documentation du langage Scheme telle que présentée dans les fichiers PDF `r5rs.pdf` et `r7rs.pdf` que nous avons téléversé sur notre répertoire GitHub. Bien que nous n'ayons pas suivi à la lettre ce que nous y avons lu sur la structure lexicale et la représentation externe du langage, cette documentation nous a grandement guidés lors de la construction de notre grammaire pour l'implantation de la fonction `read` de notre parseur.

Du côté de la gestion du répertoire GitHub, nous avons tenté d'effectuer nos changements sur des branches séparées de la branche principale afin de conserver la branche *master* dans un état fonctionnel, mais nous avons tout de même dû revenir à un état antérieur à un certain point comme nous avions tenté d'adopter une nouvelle philosophie pour le parseur qui ne s'est finalement pas révélée viable.

3 Problèmes rencontrés

3.1 Symboles `#t` et `#f`

Parmi les problèmes rencontrés, les plus notables se sont révélés être la gestion des symboles `#t` et `#f`, de la forme spéciale `let` et des listes impropres. Concernant la reconnaissance des symboles `#t` et `#f`, la difficulté fut de se rendre compte que le symbole retourné par `(string->symbol "#f")` était `|#f|` et non pas `'#f` et qu'il fallait donc explicitement retourner `'#f` en présence de la chaîne `"#f"`. Nous saurons donc à l'avenir que dans le langage Scheme, l'expression `string->symbol` ajoute une barre verticale de chaque côté du symbole résultant lorsque la chaîne à convertir débute par les caractères `#` ou `\`.

3.2 Forme spéciale `let`

La gestion de la forme spéciale `let`, nous l'affirmons avec grande conviction, fut la plus difficile à implémenter. Nous avions initialement prévu ajouter les variables qui y sont définies, accompagnées de leur valeur, dans une liste d'association représentant l'environnement, mais nous nous sommes rapidement rendu compte que cela ne permettrait pas d'accéder aux valeurs de ces variables une fois le code compilé vers l'assembleur. Ce pour quoi nous avons opté a été de garder dans l'environnement les variables déclarées au sein du `let`, mais plutôt que de les y mettre avec leurs valeurs, nous les y avons mises accompagnées du décalage à effectuer par rapport au dessous de la pile pour retrouver leur valeur sur la pile. Ainsi, lors d'un accès à ces variables, la première étape consiste à retrouver le décalage dans l'environnement d'évaluation, puis à retrouver la valeur de cette variable dans la pile. De plus, en conservant dans le registre `%rbp` la valeur de `%rsp` au premier`let` rencontré, cela nous permet d'aisément gérer un nombre arbitraire de `let` imbriqués en n'ayant aucune difficulté.

Les autres difficultés rencontrées lors de l'implémentation de la forme spéciale `let` furent celle de restaurer l'environnement à son état initial à la sortie et le fait qu'il faille évaluer chacune des expressions avant de commencer à assigner les résultats aux variables à lier. Pour ce faire, nous avons commencé par garder dans des variables (`old-env` et `old-stack`) l'état de l'environnement et du décalage par rapport au dessous de la pile avant d'étendre l'environnement avec les nouvelles affectations; cela nous permettant de pouvoir effectuer la restauration à la sortie. Ensuite, nous effectuons le calcul de l'expression dont le résultat sera affecté à la première variable. Toutefois, plutôt que d'immédiatement mettre la variable dans l'environnement (avec son décalage dans la pile), nous allons plutôt récursivement faire de même avec toutes les autres expressions à calculer pour les liaisons. C'est seulement une fois toutes ces expressions évaluées que nous regroupons toutes les variables avec leur décalage sur la pile et que nous les utilisons pour étendre l'environnement. C'est donc de cette façon que nous arrivons à respecter le comportement que doit avoir la forme spéciale `let` en Scheme.

3.3 Listes impropres

Bien que les listes propres ne nous aient causé aucun problème, il n'en fut point de même quant à la gestion des listes impropres. La structure même des listes impropres a été ce qui nous a causé le plus de problèmes, celles-ci ne devant pas débuter ni terminer par le point signifiant l'impropreté de la liste. Nous avons donc dû, au sein du parseur, ajouter deux fonctions supplémentaires pour la lecture des listes, l'une faisant office de point d'entrée afin de s'assurer que la liste ne début pas par le point, l'autre faisant office de point de sortie (nécessaire seulement dans le cas d'une liste impropre) afin de s'assurer que le point ne termine pas non plus la liste. Nous sommes toutefois satisfaits de la façon dont nous sommes arrivés à gérer cela et trouvons finalement le résultat assez propre.

4 Résultats des tests unitaires

Tous les tests fournis pour l'étape 1 fonctionnent à merveille. De plus, les tests que nous avons conçus pour tester les différentes fonctionnalités passent non seulement correctement lors du `make ut`, ils nous ont aussi été d'une grande aide tout au long du développement de notre compilateur. Comme vous pourrez le constater, certains tests unitaires échouent malgré ce qui vient d'être mentionné, mais ces échecs sont normaux au point où nous en sommes. En effet, certains d'entre eux visent à tester le parseur alors que les expressions qu'ils contiennent ne sont pas encore supportées au niveau de la génération de code, expliquant que les erreurs surviennent uniquement dans le fichier `codegen.scm` et non pas dans `parser.scm`. C'est le cas par exemple des tests suivants, qui testent le parseur sur les chaînes de caractères, la quotation et les listes, propres et impropres :

- `let-string-println.scm`
- `let-string-println-number.scm`

- `list-impropre-comp.scm`
- `list-proper1.scm`
- `list-proper2.scm`
- `list-proper-empty.scm`
- `println-let-strings.scm`
- `println-list-impropre1.scm`
- `println-list-proper1.scm`
- `println-quote.scm`
- `println-string.scm`

De plus, nous avons aussi écrit des tests dont le but est d'échouer afin de nous assurer que des expressions invalides entraînent effectivement des erreurs. C'est le cas des tests qui débutent par `crash`, tels que les suivantes :

- `crash-let.scm` → `let` sans *binding*
- `crash-println-if-let-multi.scm` → contient l'expression `(#t 156)`
- `crash-quote-double.scm` → `quote` suivi de deux arguments
- `crash-vide.scm` → liste vide sans quote (sans `'` au début)

5 Atteinte des objectifs

En conclusion, nous sommes très satisfaits du travail accompli. Au meilleur de nos connaissances, nous avons réussi à faire fonctionner complètement ce qui était demandé et avons même pris un peu d'avance sur les prochaines étapes. Nous avons hâte de voir comment gérer l'allocation dynamique de façon à pouvoir supporter le `define` et représenter les listes en mémoire.

Enfin, l'idée d'atteindre le stade où notre compilateur sera autogène nous anime et nous attendons impatiemment de voir les fonctionnalités supplémentaires que nous aurons à supporter au terme de la prochaine étape.

6 Annexes

6.1 Grammaire

Voici notre grammaire (approximative et fortement inspiré de la documentation R7RS) pour l'implantation de notre parseur :

⟨*expression*⟩ ::= ⟨*datum*⟩ | ⟨*procedure*⟩

⟨*datum*⟩ ::= ⟨*simple datum*⟩ | ⟨*list*⟩ | ⟨*abbreviation*⟩

⟨*simple datum*⟩ ::= ⟨*booleen*⟩ | ⟨*number*⟩ | ⟨*character*⟩ | ⟨*symbol*⟩ | ⟨*string*⟩

⟨*booleen*⟩ ::= `#t` | `#f`

⟨*number*⟩ ::= ⟨*explicit sign*⟩ ⟨*integer*⟩ | ⟨*integer*⟩

⟨*explicit sign*⟩ ::= `+` | `-`

⟨*integer*⟩ ::= ⟨*digit*⟩ ⟨*integer*⟩ | ⟨*digit*⟩

⟨*digit*⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

⟨*character*⟩ ::= #\⟨*any character*⟩

⟨*symbol*⟩ ::= ⟨*any character other than* ⟨*delimiter* ⟩⟩$^+$

⟨*delimiter*⟩ ::= ⟨*whitespace*⟩ | ⟨*vertical line*⟩ | `(` | `)` | `"` | `;`

⟨*whitespace*⟩ ::= `#\`space | ⟨*newline*⟩

⟨*string*⟩ ::= `"` ⟨*element string*⟩* `"`

⟨*element string*⟩ ::= ⟨*any character other than* `"` *or* `\`⟩ | `\"` | `\\`

⟨*list*⟩ ::= `(` ⟨*datum*⟩* `)` | `(` ⟨*datum*⟩$^+$. ⟨*datum*⟩ `)`

⟨*abbreviation*⟩ ::= ⟨*abbrev*⟩ ⟨*datum*⟩

⟨*abbrev*⟩ ::= `'` | `` ` `` | `,` | `,@`

⟨*procedure*⟩ ::= `(` ⟨*symbol*⟩ ⟨*datum*⟩* `)`

Figure 1: Représentation EBNF de la grammaire à supporter.