Étape 2 du projet de compilateur
Rapport

Thomas Luinaud
Francis de Ladurantaye

March 21, 2018

1 Structure logicielle

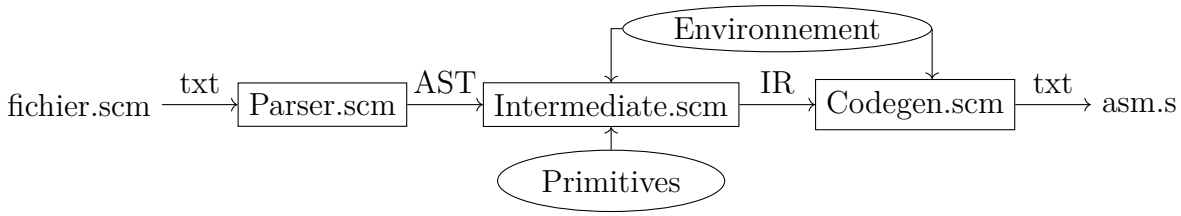

Figure 1: structure du compilateur

2 Code intermédiaire

Cette section présente les choix pour le code intermédiaire.

code intermediaire	description
`print_char`	affiche un élément
`print_num`	affiche un élément
`push_lit` *val*	ajoute un litteral sur la pile
`push_glo` *name*	ajouter la valeur d'une variable
`push_loc` *i*	empiler la valeur d'une variable locale. *i* est l'index sur la pile
`add`	depiler deux valeurs et les ajouter
`sub`	depiler deux valeurs et les soustraire
`mul`	depiler deux valeurs et les multiplier
`div`	depiler deux valeurs et les diviser
`call` *nargs*	appeler une fonction avec *nargs* argument
`ret` *i*	retourne la valeur à l'index *i* de la pile
`cmp`	dépile deux valeurs et les compare
`modulo`	dépile deux valeurs et fait le modulo
`remainder`	dépile deux valeurs et fait le remainder
`quotient`	dépile deux valeurs et fait le quotient
`lesser`	résultat de la comparaison est plus petit
`equal`	résultat de la comparaison est égal
`jmp` *étiquette*	va à la position spécifiée

3 Spécification du problème

À cette étape du projet, il nous fallait être en mesure de compiler, en plus des entiers et des booléens déjà supportés à l'étape 1, les listes, les caractères, les chaînes de caractères ainsi que les constantes littérales structurées. Par ailleurs, nous devions supporter la définition et la mutation de variables globales et permettre la déclaration et les appels de fonction (lambda-expressions inclues). Une fois en mesure de compiler ce qui précède, ces outils devaient être utilisés pour construire la liste d'opérations primitives du langage ainsi qu'une biliothèque de fonctions prédéfinies, servant principalement à intéragir avec les nouveaux types de données supportés. Enfin, nous devions ajouter une phase d'expansion de macros pour les macros `cond`, `and`, `or`, `begin`, `let*`, `letrec` et `let` nommé afin de permettre à notre compilateur de supporter des énoncés conditionnels plus complexes et la définition de variables locales.

3.1 Opérations primitives

À la fin de cette étape, en plus de ce qui était demandé à l'étape 1, il nous était donc demandé d'implanter les opérations primitives suivantes :

- (`$number?` *expr*) : test de type pour les entiers
- (`$read-char`) : lecture d'un caractère sur stdin
- (`$write-char` *expr*) : écriture d'un caractère sur stdout
- (`$integer->char` *expr*) : conversion d'entier vers caractère
- (`$char->integer` *expr*) : conversion de caractère vers entier
- (`$char?` *expr*) : test de type pour les caractères
- (`$make-string` *expr*) : création d'une chaîne de caractères ayant la longueur indiquée
- (`$string-ref` $expr_1$ $expr_2$) : extraction d'un caractère à un certain index
- (`$string-set!` $expr_1$ $expr_2$ $expr_3$) : modification d'un caractère à un certain index
- (`$string-length` *expr*) : nombre de caractères dans la chaîne de caractères
- (`$string?` *expr*) : test de type pour les chaînes de caractères
- (`$cons` $expr_1$ $expr_2$) : création d'une paire
- (`$car` *expr*) : extraction du champ car
- (`$cdr` *expr*) : extraction du champ cdr
- (`$set-car!` $expr_1$ $expr_2$) : modification du champ car
- (`$set-cdr!` $expr_1$ $expr_2$) : modification du champ cdr
- (`$pair?` *expr*) : test de type pour les paires

- (`$procedure?` *expr*) : test de type pour les fonctions
- (`$eq?` $expr_1$ $expr_2$) : test d'identité

3.2 Fonctions prédéfinies

Une fois ajouté le support des opérations précédentes, nous avions à les utiliser pour la construction de cette bibliothèque de fonctions prédéfinies :

- (`not` *expr*) : inverse booléen
- (`boolean?` *expr*) : test de type pour les booléens
- (`null?` *expr*) : test de type pour la liste vide
- (`member` $expr_1$ $expr_2$) : test d'appartenance
- (`assoc` $expr_1$ $expr_2$) : recherche dans une liste d'association
- (`append` $expr_1$ $expr_2$) : concaténation de listes
- (`reverse` *expr*) : renverser une liste
- (`length` *expr*) : longueur d'une liste
- (`map` $expr_1$ $expr_2$) : map sur les listes
- (`char=?` $expr_1$ $expr_2$) : test d'égalité de caractères
- (`char<?` $expr_1$ $expr_2$) : test $<$ sur les caractères
- (`string=?` $expr_1$ $expr_2$) : test d'égalité sur les chaîens de caractères
- (`string<?` $expr_1$ $expr_2$) : test $<$ sur les chaînes de caractères
- (`eqv?` $expr_1$ $expr_2$) : test d'équivalence
- (`equal?` $expr_1$ $expr_2$) : test d'égalité structurelle
- (`read`) : lecture d'une donnée Scheme sur stdin
- (`write` *expr*) : écriture dune donnée Scheme sur stdout

Malheureusement, comme il sera expliqué plus en détail dans les sections subséquentes, nous n'avons réussi à implémenter qu'un mince ensemble de ces opérations et fonctions.

4 Méthodologie

Comme nous savions dès le départ que l'implantation des opérations primitives et des fonctions prédéfinies requerrait le support des définitions et appels de fonctions, c'est donc à cela que nous nous sommes d'abord attaqués. Ce faisant, nous avons consacré la majeure partie de notre énergie à cette tâche afin d'éviter de devoir réécrire du code à plusieurs

reprises. Comme le support des définitions et appels de fonctions demandait de savoir gérer la conversion de fermeture adéquatement, c'est par cela que nous avons commencé.

Autrement, dans nos moments de désespoir, nous avons travaillé sur les modules ne nécessitant pas de savoir gérer les fermetures. Parmi ces modules, on compte la génération de la représentation intermédiaire, la traduction de la représentation intermédiaire en code assembleur et la phase d'expansion de macros, tâches avec lesquelles nous avons eu plus de succès.

5 Problèmes rencontrés

5.1 Conversion de fermetures

La conversion des fermetures a de loin été ce qui nous a causé le plus de problèmes. Il nous fallut un temps considérable pour commencer à cerner ce qui se passait au sein du fichier `closure-conv.scm` et bien que nous croyons plutôt bien comprendre les différentes étapes de la conversion de fermeture et leurs justifications, ce serait un euphémiste d'affirmer que nous ne serions point en mesure de le réécrire si nous avions à le faire. De plus, nous ne comprenions initialement pas comment traiter les make-closure afin d'en produire la représentation intermédiaire. Heureusement nous avons commencé à voir la lumière au bout du tunnel dans les deux derniers jours mais cela ne nous a pas laissé le temps d'implanter les opérations et fonctions demandées.

5.2 Représentation intermédiaire

La grande difficulté que nous avons eue au niveau de la représentation intermédiaire fut celle mentionnée au paragraphe précédent. Nous nous sommes posé un grand nombre de questions sur la façon de produire cette représentation intermédiaire lorsque nous tombions sur des expressoins de la forme `(define fun (lambda (x) ... ))`. Sachant qu'une telle expression ne doit pas exécuter le code de la lambda-expression, nous avions réalisé que ce n'était pas à cet endroit que nous devions produire le code de la représentation intermédiaire de cette lambda-expression. Comme il n'était pas possible de simplement ignorer le corps de la lambda-expression lors de la production de la représentation intermédiaire pour y revenir plus tard, nous avons fini par supposer qu'il fallait la produire lors de notre passage mais en la gardant en mémoire dans une sorte d'environnement pour les lambda-expressions. Ainsi, il serait possible d'y avoir accès plus tard lors de la génération de code assembleur, ce qui permettrait de les écrire à l'extérieur de la fonction `main`.

5.3 Expansion de macros

L'implantation de la phase d'expansion de macros nous a posé certains problèmes, mais bien moindres que pour la conversion des fermetures et la production de la représentation inter-

médiaire. Une des difficultés rencontrée a été de comprendre où devait avoir lieu l'expansion. Cela était dû au fait que nous avions déjà implanté la majorité des macros demandées lors de l'étape 1, mais que nous l'avions fait en utilisant `define-macro`. Nous nous demandions donc comment effectuer les substitutions une fois la phase de *parsing* terminée, et ce de façon récursive afin d'éliminer aussi les macros qui seraient apparues suite à une première substitution. Afin de pallier au problème, nous avons opté pour la réécriture des macros, cette fois-ci en faisant usage des capacités de *pattern matching* offertes par la macro `match` vue en classe. Il devenait donc très simple, en effectuant une substitution, d'appeler récursivement la fonction d'expansion de macros sur le code de remplacement.

Aussi en lien avec l'expansion des macros, il nous fut particulièrement difficile de trouver comment substituer correctement la forme *letrec*. La liaison des variables aux expressions se faisant en trois temps (création des variables, calcul de expressions et enfin liaison des résultats aux variables), cette macro dépasse largement les formes `let` et `let*` en ce qui a trait au niveau de complexité de son implémentation. Nous y sommes toutefois arrivés et en sommes très fier.

6 Résultats des tests unitaires

Comme la gestion des fermetures et l'ajout de la représentation intermédiaire ont demandé une refonte majeure de notre compilateur, le temps nous a malheureusement (grandement) manqué pour implanter ce qui était démandé. Notre compilateur se trouve présentement dans un état dans lequel la majorité des tests unitaires n'arrivent pas à compiler correctement et il est donc difficile de présenter les résultats à ces tests. Toutefois, nous pouvons affirmer que les tests simples (addition, soustraction, multiplication, quotient, modulo) fonctionnent s'ils ne contiennent qu'un seul niveau d'imbrication de `let`. En temps normal, nous devrions parvenir à passer avec succès l'ensemble des tests d'ici environ une semaine.

7 Atteinte des objectifs

Tel qu'expliqué lors des sections précédentes, la refonte majeure du compilateur nous a littéralement complètement empêché d'arriver à atteindre les objectifs associés à l'étape 2. Cependant, nous prenons cela avec philosophie comme nous savons que cette refonte était nécessaire et qu'elle ne sera pas à refaire. Nous aurions cependant préféré savoir dès la première étape qu'avoir une représentation intermédiaire est une quasi nécessité. Cela nous aurait permis de penser différemment notre compilateur dès le départ, nous évitant peut-être cette refonte complète qui a plombé notre atteinte des objectifs.

8 Annexes

8.1 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 2: Représentation EBNF de la grammaire à supporter.