Département d'informatique
et de recherche opérationnelle
Hiver 2016

Marc Feeley
bureau 3341
feeley@iro.umontreal.ca

Description du projet IFT3065-IFT6232 (Étape 2)
16 février, 2016

1 Introduction

L'étape 2 du projet consiste à étendre le compilateur que vous avez réalisé à l'étape 1 pour qu'il traite un sous-ensemble plus complet du langage de programmation Scheme. La date de remise est le lundi 14 mars à 23h59. Les objectifs spécifiques sont :

1. Ajouter le type *liste* (qui inclus les paires et la liste vide).
2. Ajouter le type *caractère*.
3. Ajouter le type *chaîne de caractères*.
4. Ajouter les constantes littérales structurées, par exemple `'(5 ("allo" #\x))`. Voir le fichier `genconst.scm` pour une approche simple d'implantation.
5. Ajouter la possibilité de définir, écrire et lire les variables globales, par exemple `(define nb 10)`, `(set! nb (+ nb 1))` et `nb`.
6. Ajouter l'appel de fonction et les lambda-expressions, par exemple `(lambda (x y) (+ (f x) (g y)))`.
7. Ajouter une bibliothèque de fonctions prédéfinies pour `append`, `reverse`, etc.
8. Ajouter une phase d'expansion des macros, et faire l'implantation des macros pour `cond`, `and`, `or`, `begin`, `let*`, `letrec` et `let` nommé.
9. Étendre votre jeu de *tests unitaires* pour vérifier le bon fonctionnement des fonctionnalités demandées.
10. Dans le cas des étudiants en IFT6232, votre implantation des lambda-expressions doit permettre les paramètres reste.

2 Opérations et fonctions primitives

La sémantique de Scheme permet de redéfinir les variables globales comme `+` et `modulo`. Cela permet par exemple de faire :

```
(set! modulo +)
(println (modulo 1 2)) ;; imprime 3
```

Pour obtenir cette sémantique, une approche simple consiste à modifier votre compilateur pour renommer les opérations primitives implantées à l'étape 1 en ajoutant un préfixe spécial (je vous suggère d'utiliser le "`$`"). Cela permettra de définir les fonctions standard dans un fichier `lib.scm` contenant des définitions comme :

```
;; Fichier "lib.scm"

(define println  (lambda (x)   ($println x)))
(define +        (lambda (x y) ($+ x y)))
(define -        (lambda (x y) ($- x y)))
(define *        (lambda (x y) ($* x y)))
(define quotient (lambda (x y) ($quotient x y)))
(define modulo   (lambda (x y) ($modulo x y)))
(define =        (lambda (x y) ($= x y)))
(define <        (lambda (x y) ($< x y)))

;; etc...
```

Le compilateur devra lire le fichier `lib.scm` et concaténer les expressions qu'il contient aux expressions provenant du fichier source à compiler. Les fonctionnalités de `lib.scm` (addition, comparaison, etc) seront donc disponibles au programme au moyen d'appels à des fonctions liées à des variables globales (comme `+`, `=`, etc) et le programme aura la possibilité de changer le contenu des variables globales `+`, `=`, etc.

On distinguera les *opérations primitives*, comme `$+` et `$modulo`, et les *fonctions primitives*, comme `+` et `modulo`. Pour toute *opération primitive* il faut définir dans `lib.scm` une *fonction primitive* correspondante de même nom, mais sans le "`$`".

3 Opérations primitives à ajouter

En plus des opérations primitives de l'étape 1 il faut implanter les 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

Cela va demander d'étendre le générateur de code et aussi étendre le fichier `lib.scm` pour implanter les fonctions primitives correspondantes.

4 Fonctions prédéfinies à ajouter

Les fonctions prédéfinies suivantes sont aussi à implanter dans le fichier `lib.scm` :

- `(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 plus-petit sur les caractères
- `(string=?` $expr_1$ $expr_2$`)` : test d'égalité des chaînes de caractères
- `(string<?` $expr_1$ $expr_2$`)` : test plus-petit 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 (vous devez traiter les symboles comme des chaînes de caractères, comme si `string->symbol` était la fonction identité)
- `(write` *expr*`)` : écriture d'une donnée Scheme sur stdout

Dans votre codage de ces fonctions, vous pouvez vous servir des opérations primitives précédentes. Pour définir les fonctions `null?` et `char=?` vous pourriez donc faire :

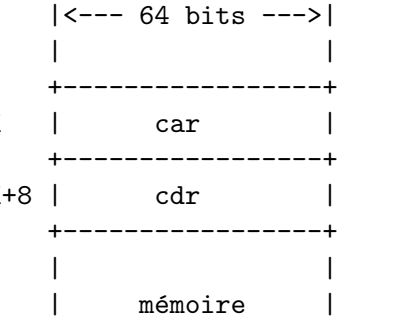
```
(define null? (lambda (x) ($eq? x '())))
(define char=? (lambda (x y) ($= ($char->integer x) ($char->integer y)))
```

5 Représentation des données Scheme

Étant donné l'ajout de nouveaux types, il faut modifier la représentation des données Scheme. Tout comme dans l'étape 1 il faut utiliser le *tagging* des valeurs et utiliser les 3 bits de poids faible. Voici une suggestion d'assignation des tags :

`000` entiers
`001` booléens et liste vide, i.e. `()`
`010` caractères
`011` chaînes de caractères
`100` *inutilisé*
`101` *inutilisé*
`110` paires
`111` fonctions

D'autre part, une *représentation indirecte* doit être utilisée pour les types chaînes de caractères, paires, et fonctions, car le reste du mot est insuffisant pour contenir toute l'information. On utilisera plutôt le reste du mot pour stocker un pointeur vers la mémoire où se trouve le reste de l'information associée à la donnée. Par exemple, pour une paire, le `car` et `cdr` de la paire seront stockés en mémoire avec 2 mots consécutifs.

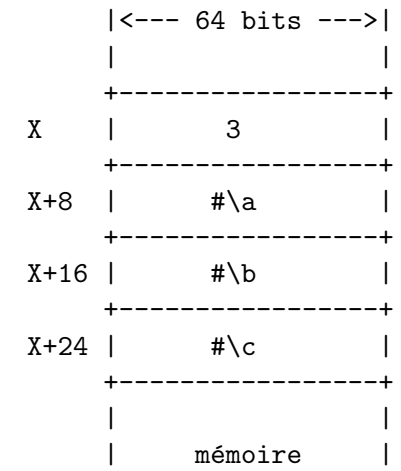


Pour simplifier la représentation et optimiser les accès mémoire, on fera toujours en sorte que l'adresse mémoire de la donnée soit un multiple de 8 (donc les données sont alignées sur un multiple de mots 64 bits). La représentation binaire de l'adresse contiendra donc toujours `000` dans les 3 bits de poids faible. On peut donc réutiliser les 3 bits de poids faible d'un mot pour stocker le tag sans qu'il y ait perte d'information. Par exemple, si une paire est stockée en mémoire à l'adresse X, alors l'encodage de cette paire sera X+6 (car 6 est le tag des paires).

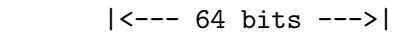
Cette approche a l'avantage qu'on peut annuler l'effet du tagging avec le mode d'adressage mémoire indirect du processeur. Par exemple, si `%rbx` contient l'encodage d'une paire, c'est-à-dire l'adresse de son champ `car` plus 6, on peut lire dans `%rax` les champs `car` et `cdr` avec les instructions `mov -6(%rbx),%rax` et `mov 2(%rbx),%rax`. Il n'y a donc pas de coût associé à l'élimination du tag. D'autres formes de tagging demanderaient de masquer les bits de tag avant de faire l'accès mémoire.

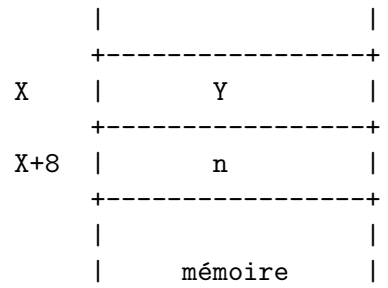
Pour la représentation des chaînes de caractères, il faut stocker en mémoire chacun des caractères et aussi, dans le premier mot, la longueur de la chaîne. Cela facilitera l'implantation de l'opération primitive `$string-length` qui n'aura qu'à lire le premier mot de la chaîne de caractères (i.e. `mov -3(%rbx),%rax` si `%rbx` contient l'encodage de la chaîne de caractères).

Je vous suggère de stocker chaque caractère dans un mot 64 bits. Cela n'utilise pas la mémoire efficacement, mais facilite l'implantation. Par exemple, la chaîne `"abc"` est représentée par :



Pour la représentation des fonctions, nous utiliserons des *fermetures* ("closures" en anglais), spécifiquement la représentation *plate* qui sera expliquée en détail en classe. Il faut stocker en mémoire un pointeur vers le code de la fonction. Après ce pointeur on stockera les *variables libres* de la fonction. Par exemple, la fonction créée par l'évaluation de la lambda-expression `(lambda (a) ($+ a n))` a `n` comme variable libre. Si Y est l'adresse où se trouve le code de la fonction, la mémoire contiendra :





6 Allocation mémoire dynamique

L'implantation des opérations primitives `$cons` et `$make-string` ainsi que les lambda-expressions, demande d'effectuer de l'allocation mémoire dynamique. Pour cela, vous devez utiliser un *tas* ("heap" en anglais) qui est l'espace mémoire dans lequel se fera l'allocation des objets de type paires, chaînes de caractères, et fermetures. Il faut faire un appel à la fonction `mmap` définie dans le fichier `mmap.s` (disponible sur le site Studium) pour demander au système d'exploitation d'allouer le tas. Utilisez un tas de longueur fixe, par exemple 100 MB. Pour cette étape du travail les données inutilisées du tas ne seront pas récupérées automatiquement (cela fera l'objet de l'étape 3) et vous n'avez pas à détecter le débordement de la mémoire.

Vous devez garder dans un registre, par exemple `%r10`, en tout temps, un pointeur vers le tas qui avance au fur et à mesure qu'on alloue des objets. C'est le *pointeur d'allocation*. Au début de l'exécution le pointeur d'allocation pointe au tout début du tas. Pour allouer un objet, on ajoute sa longueur au pointeur d'allocation.

Donc, l'allocation du tas au début du programme se fait avec les instructions :

```
mov  $100*1024*1024, %rax  # 100 MB
push %rax
call mmap         # allocation du tas

mov  %rax, %r10   # initialisation du pointeur d'allocation
```

L'appel `($cons 1 2)` peut donc se traduire par ces instructions :

```
movq $8, 0(%r10)    # initialisation du champ car
movq $16, 8(%r10)   # initialisation du champ cdr
lea  6(%r10), %rax  # rax = encodage de la paire
add  $16, %r10      # avancer le pointeur d'allocation
```

7 Variables globales

Pour implanter les variables globales, il y a plusieurs approches possibles. Le plus simple est de conserver dans un registre en tout temps un pointeur vers une zone

mémoire où se trouvent toutes les variables globales. Le compilateur n'a qu'à assigner un index unique à chaque variable et il suffira d'utiliser le mode d'adressage indirect du x86-64 avec le décallage approprié pour accéder à une variable globale spécifique. Cette zone mémoire doit être allouée au début du programme soit dans la zone `.data` du programme, sur la pile d'exécution, dans le tas ou directement avec la fonction `mmap` (comme pour l'allocation du tas).

8 Rapport

Le travail effectué à l'étape 2 doit être documenté dans un rapport contenant une description claire des objectifs visés (la spécification du problème), la méthodologie adoptée pour atteindre les objectifs, une discussion des problèmes informatiques rencontrés et l'approche utilisée pour les résoudre (algorithmes), une section présentant les résultats de tests unitaires pertinents, et une évaluation honnête de l'atteinte des objectifs (qu'est-ce qui a été accompli et les problèmes restants).

N'hésitez pas à utiliser des diagrammes et choisir quelques bons exemples pour illustrer vos propos dans le rapport. La présentation du rapport doit être sobre (pas de fontes énormes, ni des grosses tables et des diagrammes touffus!). Utilisez un logiciel de traitement de texte (Latex est fortement recommandé!). Je m'attend à des rapports de 6 à 8 pages pour IFT3065 et 8 à 10 pages pour IFT6232 (en police de 12 points, c'est-à-dire 400 à 500 mots par page).

Le rapport doit être en format PDF, et doit se nommer `etape2.pdf` à la racine de votre dépot github.