Projet d'Informatique

PHELMA 2ème année

Année universitaire 2016–2017

Programmation fonctionnelle et interpréteur
Scheme

Révisions de ce document :

Ver.	Date	Objet	Auteurs	Relecteurs
1.9	02/09/2016	Modif. incréments, sorties attendues, archive fournie, tests...	François Portet Michel Desvignes Nicolas Castagné	
1.6	01/09/2012	Modif. incréments, fermetures et méthodologie	Nicolas Castagné François Cayre	Michel Desvignes François Portet
1.0	01/09/2011	Version initiale	François Cayre	Michel Desvignes François Portet

Résumé

Le langage C est souvent qualifié d'assembleur portable, ou de langage "proche de la machine". On peut faire beaucoup de choses avec, mais certaines choses sont difficile à exprimer. En effet, la syntaxe du langage C le limite à certains égards.

Ce projet a pour but de réaliser un interpréteur pour un sous-ensemble du langage `Scheme`. Nous allons donc écrire un langage (`Scheme`) avec un autre langage (C). Le langage `Scheme` est fondamentalement différent du langage C : il n'utilise pas les mêmes paradigmes de programmation.

À la fin de ce projet, vous aurez une idée assez précise de :

— la manière dont on peut écrire un nouveau langage ;

— la puissance expressive des listes.

Mais surtout, vous aurez beaucoup développé votre culture générale en informatique en créant à partir de zéro quelque-chose qui ressemble très fort au plus puissant des langages de programmation.

Pour toute question ou éclaircissement concernant Scheme, et sauf contre-indication manifeste dans cet énoncé, vous vous référerez au R^5RS [3].

Table des matières

Chapitre 1

Présentation de `Scheme`

Différences entre C et `Scheme`

Paradigmes de programmation

Un paradigme de programmation désigne une manière de structurer la pensée humaine en vue de la résolution d'un problème par une machine. Le langage C dispose du paradigme de programmation impérative : programmer *"en impératif"*, cela veut dire que l'on spécifie les instructions à éxécuter les unes après les autres. Le langage `Scheme` dispose du paradigme de programmation fonctionnelle : lorsque l'on programme *"en fonctionnel"*, on voit la solution du problème à résoudre comme une fonction (au sens *mathématique* du terme) liant les entrées à la sortie. Cette fonction peut naturellement être définie à l'aide d'autres fonctions. Un autre paradigme connu est celui de la programmation orientée objet . Les langages les plus représentatifs du paradigme *"orienté objet"* sont C++, Java, Smalltalk et CLOS (Common LISP Object System). Un langage peut bien sûr intégrer de multiples paradigmes : C++ et Java sont impératifs et orientés objet, CLOS est fonctionnel et orienté objet. `Scheme` est en réalité un dialecte de LISP, l'un des plus anciens et très certainement le plus puissant (le plus expressif) des langages informatiques. Leurs syntaxes sont d'ailleurs quasiment identiques et extrêmement simples. Pour vous faire une idée de ce à quoi ressemble un programme `Scheme` vous pouvez vous reporter à la Figure A.1 [1].

Modèles de développement

En C, vous utilisez nécessairement le modèle de développement suivant :

1. Écriture du code source ;
2. Création d'un exécutable ;
3. Tests et retour en 1 tant que le comportement du programme n'est pas satisfaisant.

En `Scheme`, on utilise un modèle qui ressemble davantage à ceci :

1. Écriture de fonctions dans un *environnement* ;
2. Modification de l'*environnement* par l'évaluation des fonctions sur les entrées ;
3. Tests et retour en 1 tant que l'environnement n'est pas dans un *état* satisfaisant.

La notion d'environnement

En C, il n'existe pas vraiment de notion d'environnement comme en `Scheme`, juste celle de portée des variables : le lieu du code où elles sont connues. En C, la portée des variables est dépendante du bloc d'instructions dans lequel elles sont définies (ce bloc pouvant être une fonction), ou plus généralement une unité de traduction (un fichier C parmi d'autres).

En `Scheme`, un environnement contient des symboles : des noms associés à leur valeur. On peut bien sûr disposer de plusieurs environnements, qui seront alors imbriqués les uns dans les autres. La notion la plus proche en C est celle de bloc d'instructions pouvant contenir des symboles dont la portée

1. N'essayez pas de tout comprendre tout de suite, ce programme devra être compréhensible pour vous à la fin de la lecture de ce document.

est locale, en plus des symboles définis dans les blocs englobant le bloc considéré. Mais `Scheme` est beaucoup plus puissant que cela !

Lorsque l'on définit un symbole (qui peut être le nom d'un objet usuel, mais aussi d'une fonction), on associe une chaîne de caractères à une valeur. Cette opération s'appelle la création d'un lien (*binding* en anglais). Un environnement peut alors être défini comme la liste de tous les *bindings* qu'il contient.

Afin de modéliser la portée (*scope* en anglais) d'un symbole, on peut représenter les différents environnements comme étant les éléments d'une liste d'environnements. Le lien entre ces environnements s'appelle le lien de portée (*scope link* en anglais). Le premier environnement de la liste, initialisé lors du lancement de `Scheme`, et qui contient les différentes fonctions prédéfinies du langage `Scheme`, est appelé environnement de haut niveau (*top-level environment* en anglais). L'environnement de haut niveau est encore appelé niveau *meta* du langage. Il contient surtout les *formes* du langage abordées à la Sec. 1.3.

Il est également possible de définir un symbole de même nom mais de différentes natures d'un environnement sur l'autre. On dit alors qu'il se produit un phénomène dit de *masquage* : la définition la plus récente masque la plus ancienne. Nous résumons notre propos Fig. 1.1.

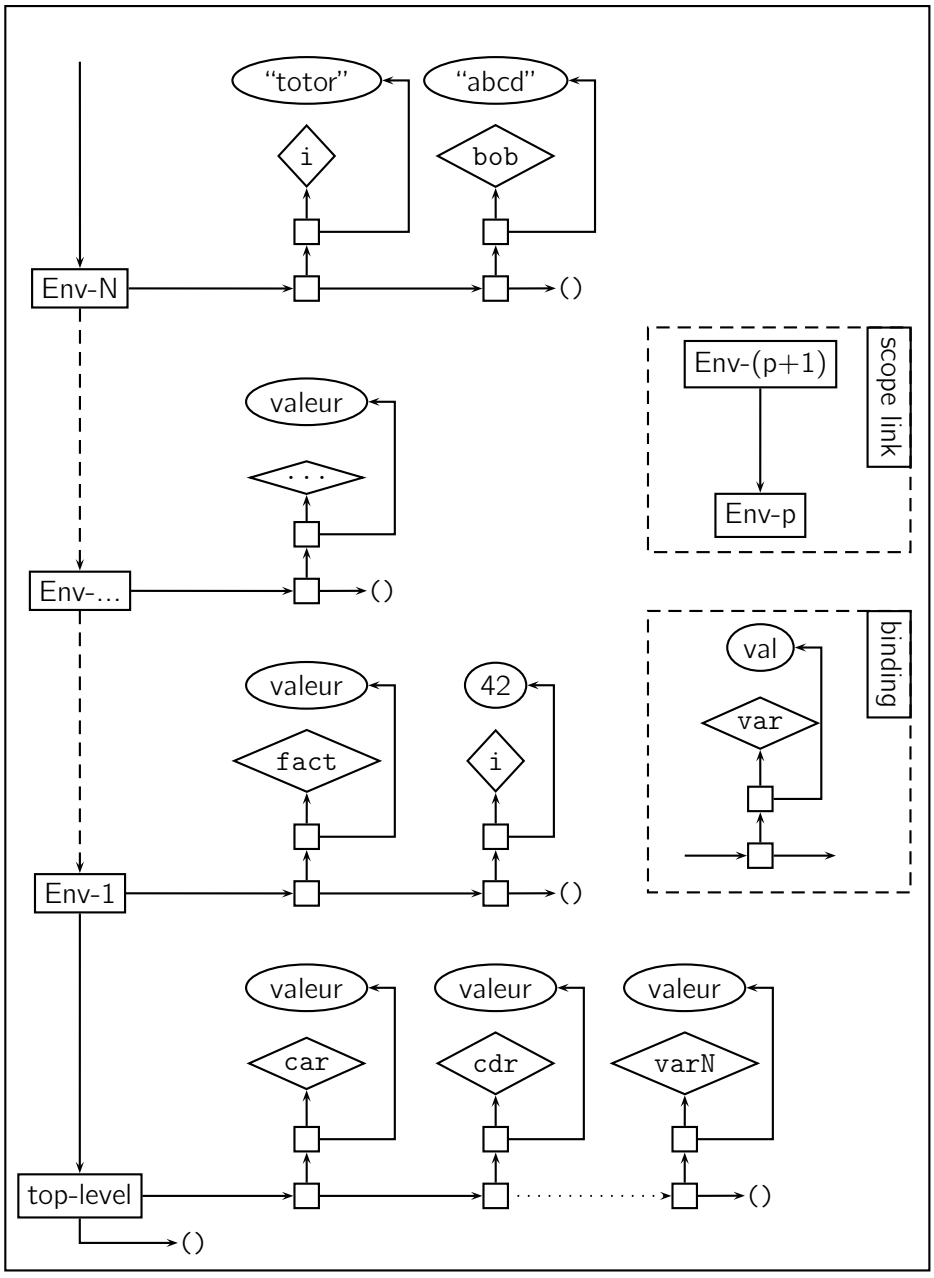

Figure 1.1 – Différents environnements en `Scheme` (les rectangles et les □ désignent des *"paires"*, voir la Ssec. 1.2.3 pour la définition d'une *paire*). Le *top-level*, ou niveau *meta* du langage, est l'environnement tel qu'initialisé au lancement de l'interpréteur. Remarquons qu'on peut redéfinir des symboles de même nom (par exemple `i`) avec des valeurs ou même des natures (variables ou fonctions) différentes dans différents environnements. Si une fonction s'exécute dans l'environnement 1, elle verra `i` comme un entier de valeur 42, alors que si elle s'exécutait dans l'environnement N, elle verrait `i` comme une chaîne de caractères de valeur "totor". On dit que la déclaration de `i` dans l'environnement N *masque* celle dans l'environnement 1.

Quelques grandes idées de la programmation fonctionnelle

Interpréteur `Scheme` et boucle REPL

C'est l'utilisateur qui doit guider les tests de mise au point des fonctions, et cela sans créer d'exécutable. Il faut donc que l'utilisateur puisse interagir avec l'environnement. En `Scheme`, on lance donc ce que l'on appelle un interpréteur : un programme qui va définir un environnement de base, dans lequel l'utilisateur va pouvoir entrer ses variables, ses symboles, ses fonctions, les évaluer, les modifier, *etc*. Un interpréteur `Scheme` fonctionne sous la forme d'une boucle infinie, qui répète sans cesse ces trois étapes :

1. Lecture de l'entrée de l'utilisateur ;
2. Évaluation de l'entrée de l'utilisateur ;
3. Affichage du résultat et retour en 1.

En anglais, on parle de boucle REPL (*read-eval-print loop*).

Un interpréteur `Scheme` peut fonctionner de façon interactive : dans un *shell*, l'utilisateur saisit une à une ses instructions `Scheme` au clavier, et obtient immédiatement le résultat de chacune d'elle à l'issue d'une itération de la boucle REPL. Un interpréteur peut aussi fonctionner sur une série d'instructions enregistrées dans un fichier texte, appelé script. On donne plus loin (Sec. 1.3) quelques exemples de sessions interactive de la boucle REPL de l'interpréteur `Scheme`. Notez qu'on pourrait, une fois que l'on est content de nos fonctions, produire un exécutable, mais ce n'est pas ce qui nous intéresse ici.

Les fonctions ne sont plus des citoyens de seconde zone

En `Scheme`, on va donc beaucoup manipuler des fonctions[2]. Idéalement, on devrait pouvoir les manipuler comme n'importe quel autre objet (nombres, caractères, *etc.*) Remarquons qu'on est tout de suite limité à cet égard en C : en C, une fonction peut au mieux renvoyer un pointeur de fonction, mais en aucun cas une fonction en tant que telle. Le C ne gère donc pas complètement, loin s'en faut, les fonctions comme des variables ordinaires. En `Scheme`, au contraire, on va pouvoir écrire des fonctions capables d'écrire d'autres fonctions, et de les renvoyer comme résultat[3]. Une fonction sera juste une autre sorte de variable, au même titre que les nombres et les caractères. Le fondement théorique de ce que l'on vient d'évoquer s'appelle le λ-calcul et est en réalité un renversement complet de la manière dont on construit les mathématiques (on les construit du point de vue des *fonctions*, donc des machines, et non plus des ensembles), cf [4] pour une introduction et beaucoup de points communs avec ce qui suit. Il se trouve qu'on peut définir les entiers naturels à l'aide d'une théorie des fonctions et, partant, toutes les mathématiques ! Le monde à l'envers...

En mathématiques, une fonction exprime essentiellement une ou plusieurs relations entre des variables muettes. Une fonction, tout comme les variables muettes qu'elle met en jeu, a un nom. Ce nom est donc relié à un symbole. Il faudra donc trouver une manière de décrire en machine ce qu'est un symbole, que ce symbole se rapporte au nom d'une fonction ou à celui d'une variable muette, *etc.* Mais en `Scheme`, on va pouvoir encore faire mieux : on va pouvoir manipuler des fonctions *anonymes* : des fonctions qui ne sont reliées à aucun nom de symbole[4] !

2. Puisque nous nous intéressons au paradigme de programmation *fonctionnelle* !
3. Vive le λ-calcul !
4. C'est pas génial le λ-calcul ?

Le problème qui se pose alors est celui du choix de la structure de données permettant de représenter une fonction en machine. Nous verrons que les listes vont jouer un grand rôle ici comme dans beaucoup d'autres aspects du langage `Scheme`.

Listes et objets

On l'aura compris, la notion de liste est absolument essentielle en `Scheme`.

En C, en première année, vous avez appris à manipuler une liste chaînée à l'aide d'une structure telle que présentée sur les Figs. 1.2–1.3.

```
typedef struct maillon_t {

  ELEMENT val;
  struct maillon_t *suiv;

} maillon, *liste;
```

Figure 1.2 – Définition d'une structure pour une liste chaînée en C (Cf. votre cours de première année).

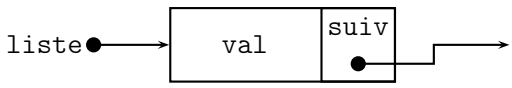

Figure 1.3 – Représentation graphique d'un maillon d'une liste chaînée en C. On fait implicitement le choix d'un stockage *interne* pour `val`, à l'intérieur de la structure de maillon.

Le champ `suiv` correspond à l'adresse de l'élément suivant dans la liste. Cependant, on peut également le voir comme l'adresse du *reste de la liste*. Et le champ `val` peut donc être vu comme le *premier* élément de la liste.

Pour gagner en généricité, la façon dont on va structurer les listes en `Scheme` est un peu différente.

En `Scheme`, on voit une liste comme étant constituée de *l'adresse* de son premier élément et de *l'adresse* du reste de la liste. Le premier élément d'une liste s'appelle le `car`[5], et le *reste* de la liste s'appelle le `cdr`[6]. Du fait de l'importance de ces deux éléments, `car` et `cdr` seront en `Scheme` aussi les noms de deux opérateurs permettant d'accéder au `car` et au `cdr` d'une paire.

De plus, on ne va plus stocker `val` (le `car` de la liste) directement, mais plutôt un pointeur vers une structure de *type* quelconque. La structure de donnée obtenue porte le nom de *paire*. Pour coder une telle paire en C, on utilise la `struct` de la Fig. 1.4 ; on se alors retrouve dans la configuration de la Fig. 1.5.

La notion de paire est, comme nous allons le voir, particulièrement malléable...

Tout d'abord, en `Scheme`, à la différence de ce que vous avez vu en C en première année, une paire peut contenir des objets de *types* différents : le `car` et le `cdr` d'une paire peuvent pointer des objets de type quelconque. Pour pouvoir coder cela en langage C, on utilisera pour les objets une structure

5. Prononcer "car".
6. Prononcer "coudeur".

```
typedef struct pair_t {

  object *car;
  object *cdr;

} *pair, *list;
```

Figure 1.4 – Définition d'une structure pour liste chaînée en C "à la `Scheme`". Il s'agit de la même structure qui décrit une *paire* en `Scheme`, mais `object` peut juste être un peu plus varié !

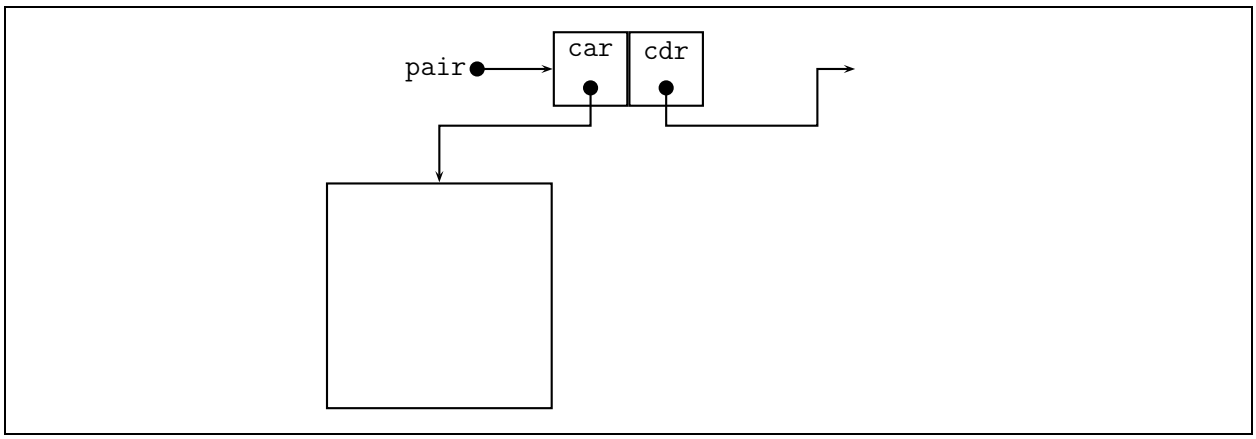

Figure 1.5 – Représentation graphique d'une *paire* en `Scheme`. On fait maintenant le choix d'un stockage *externe* pour `car` (par pointeur).

avec une `union`[7], telle que la `struct` de la Fig. 1.6. Vous remarquerez que cette `struct` définit tous les objets élémentaires de type basique dont on aura besoin (booléens, entiers, symboles, caractères et chaîne de caractères par exemple) ; elle définit également le type *paire*.

Si le `cdr` d'une paire est un objet élémentaire, on a affaire à une simple paire, ou ≪liste impure≫ à deux éléments. Une telle paire est dénotée `(obj0 . obj1)` en `Scheme`[8].

À l'inverse, si le `cdr` d'une paire est lui-même une paire, alors on a affaire à une liste chaînée à au moins deux éléments : le `car` de la première paire et le `car` de la seconde. Autrement dit, pour obtenir une liste avec de nombreux éléments, il suffit de chaîner des paires par leur `cdr`. Les éléments de la liste sont alors les `car` de toutes les paires que l'on a chaînées.

Comment dénote-t-on la fin de la liste ? Vous constaterez que dans la Fig. 1.6 on a rajouté un objet spécial de type `empty_list`. Cet objet a le même emploi que la constante `NULL` que vous utilisiez en C pour marquer la fin d'une liste. Seulement, en `Scheme`, il s'agit d'un *objet* spécial, qui sera initialisé dès la création de l'environnement de base (l'objet ≪liste vide≫ fait donc partie du niveau *meta* de `Scheme`). En `Scheme`, la liste vide sert donc, par une convention quasi-identique à celle du C, à marquer la fin d'une liste : le `cdr` de la dernière paire pointera sur `empty_list`. La liste vide se note : `()`.

En plus de `car` et `cdr`, qui permettent respectivement d'accéder au `car` et au `cdr` d'une paire, il

7. On rappelle qu'une `union` permet de voir la même variable en C sous différents types.
8. Par souci de simplicité, il ne vous est pas demandé de tenir compte de cette syntaxe dans votre implantation.

existe un troisième opérateur très important en Scheme : l'opérateur cons. On s'en sert pour *cons*truire une paire à partir de deux objets. cons est donc l'opérateur permettant de réaliser l'ajout en tête dans une liste : il suffit pour cela que le second objet passé à cons soit lui-même une liste !

Enfin, en Scheme, une paire est un type d'objet comme un autre (dans la Fig. 1.4, le champ car, de type object, peut tout aussi bien pointer un objet élémentaire qu'être *aussi* de type struct pair_t*). Une liste peut donc bien évidemment contenir d'autres listes. Pour ce faire, il faut que le car d'une des paires de la liste soit lui-même une paire et non pas un objet élémentaire.

```
typedef struct object_t {

  unsigned int type;

  union {
    int               integer;
    char*             symbol;
    char*             string;
    char              character;
    struct object_t *boolean;

    struct { struct object_t   *car;
             struct object_t   *cdr; } pair;

    struct object_t *empty_list;

  } this;

} *object;
```

Figure 1.6 – Définition d'une structure d'objet quelconque (types de base ou liste d'objets de types de base, ou liste de liste de *etc*).

.

On voit donc qu'en Scheme, un objet peut être soit un objet de n'importe quel type de base (symbole, entier, caractère, *etc.*), soit une liste d'objets de base, soit des listes de listes d'objets de base, *etc*. Par exemple, on peut très bien imaginer en Scheme la liste de la Fig. 1.7. Il s'agit d'une liste de quatre objets : une liste (contenant un nombre et une chaîne de caractères), un symbole, un caractère, et une liste (contenant un symbole, un caractère, et un nombre). Pour bien comprendre le fonctionnement des paires et des listes en Scheme, nous vous conseillons d'essayer de de représenter graphiquement cette liste à l'aide de blocs inspirés de la Fig. 1.5.

```
( ( 51 "Clyde" ) sym #\z ( f #\e 42 ) )
```

Figure 1.7 – Exemple de liste en Scheme.

Ainsi, la notion notion de paire comme base des structure de donnée en `Scheme` est particulièrement malléable. Par exemple, *une paire peut également être la brique de base pour réaliser des arbres binaires* (voir Fig. 2.1).

Fonctions et homoiconicité

Nous allons tout d'abord nous interroger sur la manière dont nous écrivons une fonction en mathématiques. Considérons par exemple la fonction f suivante :

$$f \quad : \quad \begin{array}{ccc} \mathbb{R}^3 & \to & \mathbb{R} \\ (x,y,z) & \mapsto & \sqrt{x^2+y^2+z^2} \end{array}$$

On voudra l'*évaluer* en écrivant $f(x,y,z)$. Les symboles f, x, y et z sont écrits avec la convention de faire précéder la liste des arguments d'une fonction par le nom du symbole qui la désigne (ces arguments devant être séparés par des virgules et parenthésés).

En `Scheme`, la convention est d'écrire l'évaluation d'une fonction sous la forme d'une liste dont le `car` est le nom du symbole de la fonction, et le `cdr` est la liste des arguments de la fonction. Ainsi, pour passer de l'écriture mathématique à l'écriture *à la* `Scheme`, on opère le changement suivant dans l'écriture :

$$f(x,y,z) \rightsquigarrow \texttt{(f x y z)} \tag{1.1}$$

En `Scheme`, l'évaluation d'une fonction s'écrit donc sous la forme d'une liste. Cette remarque est extrêmement importante : en effet, cela veut donc dire qu'en `Scheme` il n'existe pas de différence fondamentale entre code et données, puisque les actions (évaluations de fonctions) sont écrites sous la forme de n'importe quel autre objet. Si le symbole + désigne l'addition, alors on écrira `(+ 4 5)` pour ajouter 4 et 5. C'est de cette manière que les fonctions deviennent des citoyens de première catégorie (des objets comme les autres) en `Scheme`.

Ainsi, en `Scheme`, la manière de représenter les données est aussi la manière de représenter le code. Cette propriété fondamentale est appelée *homoiconicité*. Cela permet d'étendre assez facilement le langage une fois qu'on en a codé une amorce (un *bootstrap* en anglais). Ainsi, ce que nous aurons implanté d'ici la fin du projet fournira un *bootstrap* suffisant pour ajouter le paradigme de programmation orientée objet ou bien coder, par exemple, un compilateur, ou bien même un autre interpréteur `Scheme`, mais écrit en `Scheme` cette fois ! Les possibilités d'expression seront simplement sans limite.

Par défaut, `Scheme` va donc tenter d'évaluer la fonction dont le nom est le `car` de la liste, et les arguments le `cdr`. On a donc un problème avec une liste comme `(1 2 3)`, puisque `1` ne représente pas une fonction, mais un nombre. Il faut donc trouver un moyen de *stopper l'évaluation* lorsqu'on en a besoin. C'est le but de la *forme* `quote`. On écrira d'ailleurs indifféremment `(quote (1 2 3))` ou bien `'(1 2 3)`. La Fig. 1.8 contient un exemple de session d'une boucle REPL pour illustrer ce propos. Le langage `Scheme` propose d'autres *formes* syntaxiques canoniques, qui se rapportent à des mots-clefs fondamentaux du langage (voir Sec. 1.3.3).

Éléments de syntaxe en `Scheme`

S-expressions

En `Scheme`, on ne manipule en réalité qu'une seule sorte d'expressions : les expressions symboliques. Une expression symbolique, ou *S-expression*, est définie récursivement : une S-expression est soit un *atome*, soit une *liste* de S-expressions.

```
$ (4 5)
Error : Not a valid procedure.
$ '(4 5)
==> (4 5)
$ (quote (4 5))
==> (4 5)
$
```

Figure 1.8 – Évaluation et arrêt de l'évaluation avec `quote`.

Un *atome* peut être : la liste vide (()), un caractère, une chaîne de caractères, un booléen, un nombre entier ou un symbole[9].

Une *liste* est délimitée par des parenthèses gauche et droite, et ses éléments sont séparés par des blancs (espace, tabulation, retour chariot).

La Fig. 1.7 illustre donc une S-expression, alors que la Fig. 1.9 propose un contre-exemple.

```
( 51 ''Clyde'' ) sym #\z ( f #\e 42 )
```

Figure 1.9 – S-expressions : un contre-exemple. Ces quatre éléments ne forment pas une liste (ils ne sont pas parenthésés).

On peut décrire la syntaxe de `Scheme` à l'aide de la forme Backus-Naur (BNF en anglais). Une version simplifiée de cette syntaxe est présentée Fig. 1.10. Cette notation permet de représenter les règles de production de S-expressions valides. Une version complète de la syntaxe Scheme peut être trouvée dans la section 7.1.1 de la documentation de référence de R^{5}RSR [3].

```
liste = ''('' s-expr {s-expr} '')''

s-expr = atome
       | liste
```

Figure 1.10 – Règles de production simplifiées de S-expressions valides pour le `Scheme` en notation BNF. La première règle se lit : "Une liste est constituée d'au moins une S-expression parenthésée". La seconde se lit : "Une S-expression est soit un atome, soit une liste".

Atomes

On donne à la Fig. 1.11 les règles de production en BNF des atomes reconnus par `Scheme`.

9. Dans ce projet, on se limite volontairement à ces types simples d'atomes. La gestion des nombres réels (nombres en virgule flottante) et la gestion des nombres complexes est un bonus pour ce projet. Le canevas de code qui vous est fourni prépare l'implantation de ce bonus avec une structure `object` légèrement modifiée par rapport à celle du sujet. Notez qu'il est également tout à fait aisé d'ajouter au `Scheme` d'autres types de nombres, tels des rationnels ou des nombres exacts. Même remarque pour d'autres types de données, tels des vecteurs, des tableaux, des tables de hachage ou ce que l'on veut.

```
chiffre = 0|1|2|3|4|5|6|7|8|9

lettre = a|b|c|d|e|f|g|h|i|j|k|l|m
         | n|o|p|q|r|s|t|u|v|w|x|y|z
         | A|B|C|D|E|F|G|H|I|J|K|L|M
         | N|O|P|Q|R|S|T|U|V|W|X|Y|Z
         |\\|!|*|/     ...     ?|^|\"

booleen = ''#t''
          | ''#f''

nombre = [+|-]chiffre{chiffre}

caractere = ''#\''lettre
            | ''#\''chiffre
            | ''#\space''
            | ''#\newline''

chaine de caracteres = ''''''lettre{lettre}''''''

atome = booleen
        | nombre
        | caractere
        | chaine de caracteres
```

Figure 1.11 – Règles de production simplifiées d'atomes valides pour le `Scheme` en notation BNF. Un booléen est soit `#t` (vrai) soit `#f` (faux). Les nombres que nous manipulerons seront uniquement des entiers, positifs ou négatifs (`[x]` se lit "`x` est optionnel"). Les caractères spéciaux ont des représentations particulières en `Scheme` : une espace se note `#\space`, et un retour à la ligne se note `#\newline`. Une chaîne de caractères est délimitée par des doubles guillemets. Notez que les caractères `"` et `\` doivent être préalablement échapés par un `\`.

Formes et primitives

Une **forme** du langage `Scheme` est une S-expression dont le `car` est un des mots-clefs du langage, et qui requiert une *évaluation* particulière. Les principales formes que nous considérerons sont : `quote`, `define`, `set!`, `if`, `lambda`, `begin`, `let`. `and` et `or`.

Une **primitive** du langage `Scheme` est une S-expression dont le `car` est un des mots-clefs du langage mais qui, fondamentalement, ne requiert pas de procéder à une évaluation différente de celle d'une procédure (au contraire des formes). On choisit généralement de les implanter par solution de facilité. Les primitives que nous considérerons sont, entre autres : `cons`, `car`, `cdr`.

Pour encoder le fonctionnement d'une forme, on écrira la fonction d'évaluation (`eval`, cf. Ssec. 2.3.2) en conséquence, alors que pour encoder le fonctionnement d'une primitive, on écrira une fonction de type :

```
object *(*function)(object *);
```

Ainsi, vous devrez compléter la structure de la Fig. 1.6 en ajoutant une possibilité supplémentaire dans l'`union` (Cf. Fig. 1.12). Cette structure ressemble très fort à la structure finale que vous devrez

utiliser, mais il lui manque encore un type que nous rajouterons au moment d'aborder les agrégats (Ssec. 1.3.9).

```
typedef struct object_t {

  unsigned int type;

  union {
    int               integer;
    char*             symbol;
    char*             string;
    char              character;
    struct object_t *boolean;

    struct { struct object_t   *car;
             struct object_t   *cdr; } pair;

    struct object_t *empty_list;

    struct {
       struct object_t *
         (*function)( struct object_t * );
                                    } primitive;
  } this;

} *object;
```

Figure 1.12 – La structure d'objet quelconque mise à jour, qui permet d'encoder des primitives.

Les primitives `cons`, `car` et `cdr`

On donne dans la Fig. 1.13 un exemple de session illustrant le fonctionnement de l'opérateur `cons` et l'utilisation de la liste vide `()` pour créer des listes à partir de paires. Le nom `cons` répond à la fonction de cet opérateur : la *cons*truction de listes et de paires à partir d'autres S-expressions.

```
$ (cons 3 ())
==> (3)
$ (cons ''abc'' (cons 2 ()))
==> (''abc'' 2)
```

Figure 1.13 – Construction de paires et de listes.

De même, on donne avec la Fig. 1.14 une illustration de `car` et `cdr`. Les noms `car` et `cdr` proviennent des noms des registres de la machine utilisée pour la première implantation de LISP [10].

10. Bien qu'inventé en 1958 par John McCarthy [5], c'est Steve Russell qui a donné peu après les deux premières

```
$ (cons 1 (cons 2 (cons 3 ())))
==> (1 2 3)
$ (car (cons 1 (cons 2 (cons 3 ()))))
==> 1
$ (cdr (cons 1 (cons 2 (cons 3 ()))))
==> (2 3)
$ (car (cons 1 2))
==> 1
$ (cdr (cons 1 2))
==> 2
$
```

Figure 1.14 – `car` et `cdr`.

En pratique, on se sert beaucoup d'autres formes dérivées de `car` et `cdr` : par exemple le `car` du `cdr` se note `cadr`. De même, on va définir `caar`, `cdar`, `cddr`, `caaar`, `caadr`, `cadar`, `caddr`, `cdaar`, `cdadr`, `cddar`, `cdddr`, `caaaar`, `caaadr`, `caadar`, `caaddr`, `cadaar`, `cadadr`, `caddar`, `cadddr`, `cdaaar`, `cdaadr`, `cdadar`, `cdaddr`, `cddaar`, `cddadr`, `cdddar` et `cddddr`.

La forme `quote`

On utilise la forme `quote`, comme expliqué précédemment, pour *arrêter momentanément l'évaluation*. La Fig. 1.15 propose divers exemples. On rappelle que `(quote expression)` est équivalent à `'expression`.

```
$ (car (quote (1 2 3)))
==> 1
$ (cdr '(1 2 3))
==> (2 3)
```

Figure 1.15 – `quote` et arrêt de l'évaluation.

Les formes `define` et `set!`

On utilise la forme `define` pour introduire une variable dans l'environnement et lui donner une valeur. On utilisera la forme `set!` pour modifier la valeur associée à une variable déjà définie avec `define`. La Fig. 1.16 propose divers exemples.

implantations de LISP sur un IBM 704. Sur cette machine, les registres du processeur étaient divisés en plusieurs parties, dont deux appelées *Address* et *Decrement*. Elles étaient utilisées pour coder, en LISP, respectivement l'adresse du premier élément d'une S-expression, et l'adresse du *reste* de la S-expression. Leur nom provient respectivement de l'anglais **C***ontent of* **A***ddress part of* **R***egister*, et **C***ontent of* **D***ecrement part of* **R***egister*. Ce point d'histoire de l'informatique étant éclairci, revenons à nos symboles.

```
$ (define bob 2)
==> bob
$ bob
==> 2
$ (set! bob ''abc'')
==> bob
$ bob
==> ''abc''
```

Figure 1.16 – `define` et `set!`. On remarquera qu'on peut bien évidemment changer le type de la variable `bob` (d'un nombre vers une chaîne de caractères ici).

La forme `if`

La forme `if` sert à introduire un branchement conditionnel dans l'évaluation d'une S-expression [11]. Elle s'écrit évidemment elle-même comme une S-expression. Ce que vous écrivez en C (Fig. 1.17) devient, en `Scheme`, la forme `(if predicat consequence alternative)`. Si le `predicat` est évalué à vrai, alors le résultat sera l'évaluation de la `consequence`, sinon ce sera celle de l'`alternative`. Dans le cas où le `predicat` est évalué à faux et que l'`alternative` est absente, la forme `if` est évaluée à faux. Sur l'évaluation de la valeur de vérité en `Scheme`, voir la Ssec. 2.3.1.

```
if ( predicat ) {
  consequence();
}
else {
  alternative();
}
```

Figure 1.17 – Le bon vieux `if` du `C`.

```
$ (define x -4)
$ (if (< x 0) (- x) x )
==> 4
```

Figure 1.18 – Exemple d'utilisation de la *forme* `if` en `Scheme`. Si `x` est négatif alors le `predicat` `(< x 0)` est évalué à vrai, alors le résultat de l'évaluation du `if` est le résultat de l'évaluation de la `consequence` `(- x)`.

Note : Avec la manipulation de la mémoire (`define` et `set!`), l'introduction de la forme `if` rend notre interpréteur Turing-complet. Cela veut dire que, avec plus ou moins de facilité, nous sommes

11. Ce genre de choses au demeurant très pratiques fait pourtant figure d'hérésie dans les langages *purement* fonctionnels. En ce sens, `Scheme` ne peut pas être qualifié de langage purement fonctionnel.

désormais capables de calculer tout ce qu'un ordinateur peut espérer pouvoir jamais calculer !

La forme `begin`

La forme `begin` sert à évaluer un groupe de S-expressions dans l'ordre dans lequel elles sont écrites. C'est en quelque sorte l'équivalent d'un bloc d'instructions en C. Tout comme en C, un bloc `begin` en `Scheme` a une valeur de retour, qui est le résultat de l'évaluation de la dernière S-expression qu'il contient.

```
$ (begin 1 2 3)
==> 3
$ (begin 3 (+ 4 5))
==> 9
```

Figure 1.19 – Exemples de comportement de la forme `begin` : le résultat est l'évaluation de la dernière S-expression.

La forme `begin` est particulièrement utile lorsque l'on souhaite enchaîner plusieurs opérations séquentiellement dans une évaluation. La figure 1.20 illustre l'utilisation de `begin` dans une forme `if`.

```
$ (define x 8)
$       (if (> x 5) (begin (set! x 5) (* x x) ) (- x 5) )
==> 25
```

Figure 1.20 – Exemple d'utilisation de la *forme* `begin` en `Scheme`. Si `x` est supérieur à 5 alors le `predicat` `(> x 5)` est évalué à vrai, alors la `consequence` affecte 5 à `x` (`(set! x 5)`) puis évalue le carré de `x` (`(* x x)`). C'est cette valeur que prend l'évaluation du `begin` et par conséquent du `if`.

Note : On peut également voir la forme `begin` comme l'embryon qui nous permettra de coder les agrégats et les procédures (cf. Ssec. 1.3.9, 1.3.10).

Fonctions et notion de fermeture

On sent, avec la forme `begin`, que l'on n'est pas loin de la notion de fonction. Et, de fait, le *corps* d'une fonction (ce qu'elle doit effectuer) sera implanté sous cette forme. Mais il y a nettement plus problématique...

Si les fonctions doivent être traitées comme n'importe quel autre type de variable, cela implique en particulier que l'on puisse les renvoyer comme valeur de retour d'une autre fonction. Or, imaginons que l'on définisse la fonction `toto` à l'intérieur de la fonction `tata`, et que la fonction `toto` utilise les valeurs des variables ou des paramètres de `tata`, et que la fonction `toto` soit retournée comme valeur de retour. Un problème surgit immédiatement : à n'importe quelle évaluation de `toto`, on doit pouvoir accéder aux variables et/ou aux paramètres de `tata`[12]. La solution est donc d'associer à la structure décrivant une fonction, l'environnement dans lequel elle doit être évaluée.

12. Ce problème ne se pose pas en C, puisque, peu ou prou, les fonctions ne seront *in fine* que des adresses auxquelles se brancher dans une section de code binaire – et les variables locales ont une durée de vie limitée au bloc dans lequel elles sont définies. En Scheme, ce ne peut pas être le cas : la durée de vie des variables locales ne doit pas dépendre du lieu où elles sont définies. On a alors besoin de définir plus finement la structure qui va encoder une fonction (fermeture),

Presque formellement, on pourrait définir une fermeture comme une fonction capable d'accéder aux paramètres et aux variables locales des fonctions dans lesquelles elle est définie.

Nous définirons donc en `Scheme` une fonction comme étant une fermeture. La structure de données associée s'appellera un *agrégat*. Par agrégat, on désigne une fonction *sans son nom*. Un agrégat (*compound* en anglais) est formé de trois éléments :

— le corps, soit la liste des instructions à évaluer ;
— les paramètres sur lesquels évaluer le corps de l'agrégat ;
— l'environnement dans lequel évaluer l'agrégat.

Dans la structure de la Fig. 1.6, vous devrez rajouter de quoi coder ce type particulier. On pensera naturellement à utiliser le type de la Fig. 1.21.

```
struct { struct object_t *parms;
         struct object_t *body;
         struct object_t *envt;  } compound;
```

Figure 1.21 – Le type *compound* permettant de décrire un agrégat.

La forme `lambda`

La forme `lambda` est un peu particulière. Elle sert à définir des fonctions anonymes, que l'on peut, par exemple, définir et utiliser pendant que l'on est en train de faire autre chose. Il existe bien évidemment un lien direct avec la notion d'agrégat vue à la section précédente. On utilise la forme `lambda` lorsque l'on a besoin d'une fonction qu'on ne réutilisera pas et qui servira momentanément, ou bien pour créer un état encapsulé [13]. Pour appliquer l'agrégat anonyme à un ou plusieurs paramètres, il suffit de les faire suivre la définition de l'agrégat. On verra à la Sec. 1.3.11 comment utiliser `define` pour transformer un agrégat (anonyme) en une fonction (qui deviendra une variable identifiée par son nom). Par ailleurs, nous vous conseillons de lire ces sections ensemble afin de mieux comprendre les subtilités en jeu.

`define`, reloaded !

Vous devrez modifier la forme `define` pour pouvoir gérer le nommage des agrégats en plus de celui des variables. C'est l'utilisation de la forme `define` qui permettra de nommer un agrégat et d'en faire une procédure à part entière (en associant un agrégat à un nom de symbole !). Nous verrons deux manières de faire cela : la premiére étant évidente mais lourde à utiliser (Fig. 1.23), et la seconde étant un raccourci plus intuitif (Fig. 1.24).

Dans la Fig. 1.23, on sent tout autant la logique que la lourdeur. Aussi, on devra également pouvoir accepter la syntaxe suivante : la liste suivant le mot-clef `define` contiendra le nom de la procédure (son `car`), et ensuite ses arguments (le `cdr` de cette liste), voir Fig. 1.24.

Lorsque vous aurez implanté cette partie, vous serez capable d'utiliser des procédures en paramètres d'autres procédures (voir Fig. 1.25).

et qui pourra donc être renvoyée comme valeur de retour. Cet accès aux variables locales d'une fonction par une fonction qui est définie à l'intérieur de la premiére, mène à la notion d'*état encapsulé* – qui peut être une manière, parmi plusieurs autres, d'implanter des éléments de programmation orientée objet.

13. L'état d'une variable est dit encapsulé lorsque le seul moyen de le modifier est de passer par une fonction.

```
$ (lambda (x) (* x 2))
==> #<procedure>
$ ((lambda (x) (* x 2)) 4)
==> 8
$ x
Error : Unbound variable 'x'.
$ lambda
Error : Unbound variable 'lambda'.
```

Figure 1.22 – Quelques exemples d'utilisation de la forme `lambda`. Notez que la première *définit* une fonction (anonyme, et donc pas réutilisable), alors que la seconde, en plus de la définir, procède à son *évaluation* sur l'argument `x` qui prendra donc la valeur `4` pour cette évaluation. Dans le second cas, l'environnement est étendu *avant* de procéder à l'évaluation, soit au moment de la définition de la forme (ici uniquement pour tenir compte de l'argument `x`).

```
$ (define ma-procedure (lambda (x) (* x 2)))
$ ma-procedure
==> #<procedure>
$ (ma-procedure 4)
==> 8
$ (ma-procedure (ma-procedure 4))
==> 16
```

Figure 1.23 – Exemple de définition de procédure (première manière). On associe un agrégat à un nom (`ma-procedure`). La variable ainsi nommée est donc de type *fonction prenant un seul paramètre*.

.

```
$ (define (ma-procedure-2 x)
  (* x 2))
$ ma-procedure-2
==> #<procedure>
$ (ma-procedure-2 4)
==> 8
$ (ma-procedure-2 (ma-procedure-2 4))
==> 16
```

Figure 1.24 – Exemple de définition de procédure (seconde manière). On choisit une syntaxe plus intuitive pour la définition de fonctions.

.

Un exemple plus complet

On se souvient que la forme `lambda` étend l'environnement courant au moment de la définition, en créant un environnement dans lequel le corps de l'agrégat s'exécutera. On pourra ainsi définir des états encapsulés pour des variables. Dans l'exemple de la Fig. 1.26, on montre comment utiliser la forme `lambda` pour créer un compteur.

L'explication du code de la Fig. 1.26 est la suivante :

```
$ (define (mul-by-2 x)
    (* 2 x))
$ (define (map proc items)
    (if (null? items)
        '()
        (cons (proc (car items))
              (map proc (cdr items)))))
$ mul-by-2
==> #<procedure>
$ (map mul-by-2 '(0 1 2 3))
==> (0 2 4 6)
```

Figure 1.25 – Exemples de définition de procédures. Ici, la procédure `map` permet d'appliquer une autre procédure à une liste d'arguments (`map` prend bien une autre procédure, `mul-by-2` en paramètre). La procédure `mul-by-2`, servant à doubler la valeur d'un entier, est appliquée à une liste d'entiers. Dans cet exemple, `null?` est un prédicat servant à détecter si `items` est la liste vide.

```
$ (define count
    ((lambda (total)
       (lambda (increment)
         (set! total (+ total increment))
         total))
     0))
$ (count 3)
==> 3
$ (count 5)
==> 8
$ total
Error : Unbound variable 'total'.
```

Figure 1.26 – La forme `lambda` étend l'environnement courant (on utilise cette propriété pour la variable `total`). Cela permet de ne pouvoir modifier cette valeur qu'à travers l'évaluation de la fonction `count` (l'état de `total` est dit *encapsulé*).

- La fonction `count` est définie (première manière) comme le résultat de l'*évaluation* (d'où les deux parenthèses ouvrantes devant `lambda` : la première pour forcer l'évaluation, la suivante pour la définition) de la première forme `lambda` ;
- Le corps de la première forme `lambda` est défini (première manière également) par une seconde forme `lambda` de paramètre `increment`[14] ;
- Ainsi, la variable `count` est-elle définie comme l'évaluation *in fine* du corps de la seconde forme `lambda` sur deux paramètres : le premier (`total`) appartenant à la première forme `lambda`, et le second (`increment`) appartenant à la seconde forme `lambda` ;
- Au moment de l'*évaluation* de `count`, l'interpréteur devra trouver les valeurs des arguments ainsi : la valeur du premier argument (`total`) est connue car mise à 0 au moment de la

14. On voit clairement ici à l'œuvre la notion de fermeture : la seconde forme `lambda` accède au paramètre `total` de la première (la raison pour laquelle `total` doit survivre à l'évaluation de la seconde forme `lambda`)

définition du corps de la fonction, par contre le second argument est manquant, et devra donc être passé en paramètre par l'utilisateur [15].

15. Formellement, `count` est donc du type *fonction prenant un argument* (`increment`), et puisque l'environnement est étendu au moment de la définition, la valeur de `total` est mémorisée d'une évaluation sur l'autre. En effet, lors de la définition de `count`, l'environnement est étendu pour créer le paramètre `total` et lui donner sa valeur (`0`), mais lors des évaluations ultérieures, qui ne procèdent pas à d'autres extensions d'environnement puisque ce ne sont pas des définitions, ce sera donc la valeur courante du paramètre `total` qui sera utilisée.

Chapitre 2

Mécanismes de l'interpréteur `Scheme`

La boucle REPL

Lorsqu'on lance l'interpréteur `Scheme` en mode interactif, il présente une invite destinée à recueillir une S-expression entrée par l'utilisateur. Cette S-expression est ensuite évaluée. Le résultat de l'évaluation de la S-expression de l'utilisateur est une autre S-expression, qu'il s'agit ensuite d'afficher. Et on continue ainsi indéfiniment.

Décomposition analytique (read())

Le but de la décomposition analytique d'une entrée est, à partir des caractères ASCII représentant une entrée, d'en déduire la représentation qui sera utilisée en interne par l'interpréteur pour l'évaluation. Le fait d'avoir choisi une syntaxe aussi simple que celle de `Scheme` autorise une implantation "à la main" [1]. La puissance de `Scheme` réside dans le fait que ses structures internes sont exactement celles qui régissent l'écriture d'une S-expression. Une telle décomposition analytique contient deux étapes : l'analyse lexicale, et l'analyse grammaticale [1].

Brèves généralités

Analyse lexicale

Le but de l'analyse lexicale est, en partant de l'entrée de l'utilisateur, de constituer des lexèmes. Les lexèmes sont les unités lexicales du langage. Les lexèmes peuvent être, en `Scheme` : la liste vide, des booléens, des nombres, des caractères, des chaînes de caractères, ou des symboles.

Analyse grammaticale

Le but de l'analyse grammaticale est de s'assurer que la structure des différents lexèmes donnés par l'utilisateur correspond bien à la syntaxe du langage. Pour notre interpréteur `Scheme`, on s'assurera donc que les lexèmes décrivent une S-expression correctement formée (cf. Fig. 1.10).

Arbre syntaxique

La classe de grammaires qui nous intéresse (et qui inclut celle de `Scheme`) permet de représenter n'importe quelle S-expression sous forme d'un *arbre syntaxique*. L'arbre syntaxique est le résultat des analyses lexicale et grammaticale. Un analyseur syntaxique n'est pas tenu de toujours construire un tel arbre, mais il le fait pourtant dans la grande majorité des cas et, pour nous, cela sera vital car l'arbre syntaxique sera en réalité la structure interne qui sera manipulée par notre interpréteur. On représente sur la Fig. 2.1 l'arbre syntaxique correspondant à la S-expression de la Fig. 1.7.

1. Un programme écrit, par exemple, en C, est bien plus difficile à décomposer analytiquement. Corollaire : c'est la raison pour laquelle on ne vous demande pas d'écrire à la main en C un compilateur pour le C...

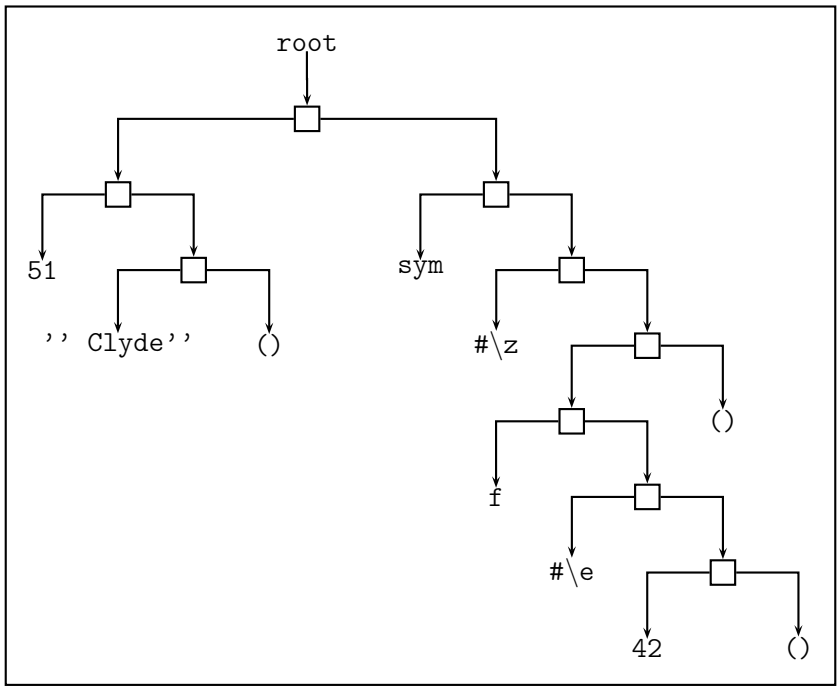

Figure 2.1 – Arbre syntaxique de la S-expression de la Fig. 1.7. Les □ désignent des paires (`car` à gauche et `cdr` à droite). Notez que nous n'avons pas représenté le *type* de chaque objet, mais en réalité, lorsque l'arbre syntaxique est contruit, le type de chaque objet est connu (chaque nœud de l'arbre syntaxique est bien évidemment de type `object*`). C'est parce que la S-expression est correcte que l'on peut construire l'arbre (et donc parce que l'expression est grammaticalement correcte). C'est ici que la remarque concernant les arbres binaires (Sec. 1.2.3) prend tout son sens.

Approche proposée : analyseur descendant récursif

Dans le cas le plus général, les étapes d'analyses lexicale et grammaticale sont assez difficiles à mettre en œuvre. On a alors recours à des outils dédiés : pour réaliser un analyseur lexical, on utilisera par exemple `lex` ou `flex`, alors que pour créer un analyseur grammatical, on utilisera par exemple `yacc` ou `bison`.

Toutefois, dans le cas de la syntaxe extrêmement simple de `Scheme`, il est possible de réaliser *à la main* et *en même temps* les analyses lexicale et grammaticale. Pour cela, on va réaliser un *analyseur descendant récursif* (ADR). Un ADR requiert deux fonctions mutuellement récursives : l'une pour gérer les lexèmes rencontrés, et l'autre pour s'assurer qu'ils forment bien une S-expression valide (et créer l'arbre syntaxique en mémoire au fur et à mesure).

Structure de l'ADR

L'analyse syntaxique effectuée par un ADR commence impérativement par la reconnaissance de l'axiome du langage.

Formellement (cf. la grammaire en BNF de `Scheme` sur la Fig. 1.10), un `booleen`, un `nombre`, un `caractere`, une `chaine de caractere`, une `liste` ou une `paire` sont appelés les éléments **non-terminaux** du langage `Scheme`. De plus, parmi ces non-terminaux, on distingue les **atomes** du langage et les paires (ou listes) - une paire étant constituée de deux non-terminaux.

Pour reconnaître à quel élément du langage on a affaire, on peut remarquer qu'il suffit, en `Scheme`,

d'examiner le caractère suivant dans le flux d'entrée.

En effet, si on lit une parenthèse ouvrante (`(`), alors on doit lire une liste ou une paire (sauf si le caractère immédiatement suivant est une parenthèse fermante (`)`), auquel cas on est en train de lire la liste vide).

Et, pour les atomes, si on lit :

— un `+`, un `-` (non suivis d'un blanc) ou un `chiffre`, alors on doit lire un nombre ;

— un guillemet (`''`), alors on doit lire une chaîne de caractères (qui se termine au prochain guillemet) ;

— un dièse (`#`), alors on doit examiner le caractère suivant :

  — si c'est un *backslash* (\), alors on est en train de lire un caractère (on fera attention à détecter correctement les caractères spéciaux comme `#\newline` ou `#\space`) ;

  — si c'est un `t` ou un `f`, alors on est en train de lire un booléen ;

— tout autre caractère alphanumérique, alors on est en train de lire un symbole [2].

La détection du type de non-terminal auquel on a affaire est donc relativement simple...

Fonctionnement de la fonction `read()` et des fonctions associées ; récursivité mutuelle

On pourra par exemple utiliser le prototype suivant pour la fonction `read()` de construction d'un objet à partir d'une chaîne (c'est à dire la fonction qui implantera l'ADR) :

```
object *read( char* input, int* );
```

Cette fonction prend en paramètre la chaîne de caractères contenant le code `Scheme` à traiter [3] et renvoit la racine de l'arbre syntaxique (`root` dans la Fig. 2.1).

La fonction `read()` devra d'abord déterminer si elle a affaire à un atome ou à une liste.

Si `read()` a affaire à un atome, elle appellera une fonction dont le rôle sera de reconnaître l'atome, de le lire et de renvoyer l'`object` correspondant. On appellera `read_atom()` cette fonction.

Si à l'inverse `read()` a détecté qu'elle a affaire à une paire (ou liste), elle appellera une autre fonction qui se chargera spécifiquement de construire des paires, et aura donc à rappeler `read()` au moment de créer les non-terminaux correspondant au `car()` et au `cdr()` de la paire. On appellera cette fonction `read_pair()`.

Les fonctions `read()` et `read_pair()` s'appellent donc l'une l'autre. Ce sont les deux fonctions mutuellement récursive que nous évoquions plus haut.

Au sujet de la création des non-terminaux

Nous précisons ci-dessous quelques aspects liés à la création des non-terminaux en mémoire par l'ADR.

La liste vide

On ne définira qu'un seul objet de type liste vide, qui sera initialisé au moment du lancement de l'interpréteur `Scheme`. C'est l'adresse de cet objet que l'on renverra lorsque l'analyseur détectera une liste vide dans l'entrée de l'utilisateur.

2. Voir plus loin pour la gestion des formes.

3. Notez que cette chaîne aura été préalablement construite en mémoire à partir de l'entrée de l'utilisateur, soit depuis le clavier, soit depuis un fichier texte contenant du code `Scheme`, au moyen d'une fonction dont le code vous sera fourni.

Booléens

En pratique, on veillera à définir uniquement deux objets de type booléen (cf. Fig. 1.6), un pour `#t`, et l'autre pour `#f`. Lorsqu'une S-expression contient ou renvoie `#t` (resp. `#f`), on utilisera ou on renverra l'adresse de `#t` (resp. `#f`).

Nombres, (chaînes de) caractères

Pour les nombres, les caractères et les chaînes de caractères, on peut être confronté au problème suivant : en admettant que l'utilisateur entre la S-expression `(+ 3 3)`, devons-nous allouer en mémoire deux objets de type nombre et de même valeur (3), ou bien n'en allouer qu'un seul en le référençant deux fois ? Nous laissons ce choix à votre discrétion, tout en soulignant que des allocations multiples sont probablement la solution la plus simple à mettre en œuvre (on aborde plus avant les aspects de gestion de la mémoire dans la Sec. 3.3.4).

Symboles

Les symboles, par contre, ne peuvent être alloués qu'une seule fois. On les rangera donc dans une table des symboles. Si un symbole est détecté dans l'entrée de l'utilisateur, on vérifiera d'abord qu'il n'existe pas déjà dans la table des symboles. S'il n'y figure pas encore, alors on le créera et on renverra l'adresse du nouvel objet créé, sinon on renverra son adresse déjà existante.

En pratique, il existe deux manières de réaliser une table des symboles. La manière simple consiste à les ranger dans une liste. La manière la plus efficace consiste à utiliser une table de hachage. Dans le cadre de ce projet, il vous est fortement conseillé d'utiliser des listes `Scheme` pour construire votre table de symboles. L'optimisation consistant à implanter une table de hashage est un bonus.

Paires (listes)

Lorsque l'ADR rencontrera une parenthèse ouvrante, il devra d'abord vérifier qu'il ne s'agit pas du début de la liste vide. Si tel n'est pas le cas, l'ADR va créer une paire destinée à initier la liste des éléments contenus dans la S-expression.

Formes

Comme expliqué plus haut, les formes sont des S-expressions dont le `car` est un des mots réservés du langage. Ces mots réservés sont des symboles que l'on créera au moment de l'initialisation de l'interpréteur `Scheme`. On prendra soin de créer une fonction destinée à vérifier si le `car` d'une S-expression est un des mots réservés du langage. Cette fonction vous sera particulièrement utile au moment de coder l'étape d'évaluation.

Dans le cas particulier de la forme abrégée de `quote` (on rappelle que `(quote a)` et `'a` sont la même chose), on veillera à créer explicitement en mémoire une S-expression contenant le symbole `quote`. En d'autres termes, la forme abrégée de `quote` n'est qu'une facilité donnée à l'utilisateur, mais n'est pas utilisée dans la représentation interne de la S-expression entrée par l'utilisateur.

Évaluation (eval())

En `Scheme`, les objets ne s'évaluent pas de la même manière suivant leur type. Certains objets sont auto-évaluants : ils sont eux-mêmes le résultat de leur évaluation. C'est le cas notamment des booléens, des nombres, des caractères ou des chaînes de caractères.

D'autres types d'objets, par contre, requièrent des processus d'évaluation variés. C'est le cas des symboles, des formes et des listes.

Objets auto-évaluants

Booléens et valeurs de vérité en `Scheme`

Les booléens sont auto-évaluants. Nous avons vu (cf. Ssec. 1.3.2) que la valeur vrai se note `#t`, et que la valeur faux se note `#f`. En `Scheme`, tout est vrai, à l'exception de `#f`. Il s'agit d'une autre différence avec le C : en C, 0 est faux, alors qu'il est vrai en `Scheme`.

Nombres, (chaînes de) caractères

Les nombres, les caractères et les chaînes de caractères sont auto-évaluants.

Objets non auto-évaluants

Symboles

Les symboles sont compris comme désignant des variables en `Scheme`. Ils s'évaluent en la valeur de la variable qu'ils désignent. On fera notamment attention à respecter le principe de masquage évoqué à la Sec. 1.1.3. La seule exception concerne les formes. Si un symbole est le `car` d'une S-expression et qu'il désigne une forme du langage, alors on exécute ce que prescrit la forme en question.

Paires (listes)

Une S-expression, paire ou liste, s'évalue comme une fonction dont le `car` désigne une variable de type fonction, appliquée au `cdr` de la S-expression.

Pour le cas où le `car` de la S-expression ne serait ni un mot réservé du langage, ni un symbole ou ni un symbole se rapportant à une fonction, on renverra la liste vide `()`. Un message d'erreur indiquera tout de même pourquoi l'évaluation a échoué.

Formes

Les formes nécessitent pour chacune d'elle une adaptation de l'étape d'évaluation. On se référera à la Sec. 1.3.3 pour les détails du comportement qu'elles doivent chacune respecter. Par exemple, il est évident que la forme `if` rend la fonction d'évaluation récursive. En effet, son évaluation requiert l'évaluation de la valeur de vérité de son prédicat.

Affichage du résultat (print())

L'étape d'affichage est en réalité une sorte de mirroir de l'étape de lecture. Elle aussi va utiliser deux fonctions mutuellement récursives : l'une pour l'affichage des objets simples (correspondant aux non-terminaux à l'exception des paires), et l'autre gouvernant l'affichage d'une paire.

La Figure 2.2 illustre l'affichage attendu de plusieurs expressions. Le résultat de l'évaluation de chaque S-expression devra être affiché dans la sortie standard `stdout` du programme sur UNE seule ligne. Cette ligne commencera obligatoirement par `==>` (notez l'espace après ">"). Les parenthèses d'une liste seront affichées sans espace et les éléments d'une liste seront séparés d'un seul espace. Si un ensemble de paires imbriquées n'est pas terminé par `()` alors un point sera affiché avant le dernier

élément (convention Scheme). Pour les objets dont l'affichage n'a aucun sens, on renverra un message canonique. Par exemple, puisqu'afficher une fonction n'a pas beaucoup de sens, on se contentera de renvoyer le message `#<procedure>`.

Enfin, puisque l'erreur est humaine, les erreurs des programmes Scheme devront être signalées à l'utilisateur. Dans le mode interactif, ceci se traduira par un message à l'écran (sans `==>` puisqu'il ne s'agit pas du résultat d'une évaluation, et envoyé sur le flux d'erreur standard `stderr` du programme) alors que dans le mode *script*, en plus d'écrire le message d'erreur dans `stderr`, le programme arrêtera l'exécution et renverra un code erreur.

Il est important de respecter strictement ces conventions d'affichage pour la mise en œuvre des tests unitaires. Le non respect de celles-ci sera sanctionné comme il se doit.

```
$ 1
==> 1
$ '100
==> 100
$ (quote abc)
==> abc
$ "␣\"mon\"␣\\repertoire"
==> "␣\"mon\"␣\\repertoire"
$ #\&
==> #\&
$ ( quote (    if    ( >    a 5)
        (  + 9              8)  ))
==> (if (> a 5) (+ 9 8))
$ (> 3 2)
==> #t
$ (define a 9)  ; valeur de retour non definie dans la doc
$ (set! a 8)    ; valeur de retour non definie dans la doc
$ a
==> 8
$ (cons 1 2) ; paires imbriquees
==> (1 . 2)
$ (cons 1 (cons () (cons 3 4))) ; paires imbriquees
==> (1 () 3 . 4)
$ (cons 1 (cons () (cons 3 (cons 4 () )))) ; liste
==> (1 () 3 4)
$ (lambda (x) (+ x x))
==> #<procedure>
$ ((lambda (x) (+ x x)) 2)
==> 4
$ (define power2 (lambda (x) (* x x)));valeur de retour non
                                      ; definie dans la doc
$ power2
==> #<procedure>
$ (power2 2)
==> 4
$ (quote a b c)
Erreur - quote ne prend qu un argument (ou autre message)
$ (5)
Erreur - l'expression "(5)" n'est pas valide (ou autre message)
$ (quote 5) )
Erreur - parenthese ")" inattendue (ou autre message)
```

Figure 2.2 – Affichage du résultat de l'évaluation de différentes expressions. Le résultat d'une évaluation sera envoyé sur le flux `stdout`. Les messages d'erreur seront envoyés sur le flux d'erreur standard `stderr`.

Chapitre 3

Méthode de travail

On discute ici de l'organisation de votre travail durant le projet `Scheme`.

Méthode de développement

Développer un logiciel tel qu'un interpréteur `Scheme` demande de travailler avec méthode pour parvenir efficacement à un code qui, par exemple, réponde au problème, soit robuste (absence de plantage), fiable (absence d'erreur, ou gestion appropriée des erreurs), clair et lisible, maintenable (facilité de reprise et d'évolution du code), etc.

La mise au point de méthodologies de développement est l'une des activités du "Génie Logiciel", une branche de l'informatique. De nombreuses méthodologies existent pour organiser le développement logiciel. Chacune définit par exemple le cycle de vie qu'est censé suivre le logiciel, les rôles des intervenants, les documents intermédiaires à produire, etc.

A l'occasion du projet, vous expérimenterez non pas une méthodologie complète - ce serait trop lourd - mais deux méthodes, ou moyens méthodologiques, qui sont promus par plusieurs méthodologies : le *développement incrémental* et le *développement piloté par les tests*.

Notion de cycle de développement ; développement incrémental

La notion de cycle de vie du logiciel correspond à la façon dont le code est construit au fur et à mesure du temps, depuis la rédaction des spécifications jusqu'à la livraison du logiciel, en passant par l'analyse, l'implantation, les tests, etc. Plusieurs cycles de vie sont possibles.

Dans le cadre du projet, vous adopterez un *cycle incrémental* - appelé également *cycle en spirale*. Il s'agit de découper les fonctionnalités du logiciel de telle sorte qu'il soit construit progressivement, en plusieurs étapes. A l'issue de chaque étape, le logiciel ne couvre pas toutes les fonctionnalités attendues, mais doit être fonctionnel - et testable - pour la sous partie des fonctionnalités réalisées jusqu'ici.

Les avantages du développement incrémental dans le cadre du projet sont nombreux. Par exemple, les risques sont réduits : au lieu de ne pouvoir tester votre logiciel qu'à la fin du projet et de risquer que rien ne marche, vous aurez à chaque étape quelque chose de partiel, certes, mais qui fonctionne. Incidemment, cela tend à garantir une plus grande implication dans l'activité de développement, puisqu'on voit la chose - le logiciel - se construire au fur et à mesure.

A l'inverse, le développement incrémental peut nécessiter une plus grande dextérité : durant un incrément, il peut être nécessaire de développer une "fausse" partie du logiciel pour "faire comme si" cette partie existait... Puis être capable de la remplacer intégralement lors de l'incrément suivant. Une autre des difficultés du développement incrémental est la définition des incréments et de l'ordre de leur réalisation - c'est-à-dire du découpage temporel de l'activité de développement. Fort heureusement pour vous, le découpage est déjà défini pour le projet !

Vous réaliserez votre interpréteur `Scheme` en quatre incréments. La partie 4 détaille les quatre incréments à réaliser ; vous vous y référerez tout au long du projet.

Développement piloté par les tests

Pour développer un logiciel, il est possible de travailler très précisément ses spécifications, de s'en imprégner, de beaucoup réfléchir, de développer... Et de ne tester le logiciel qu'à la fin, lorsqu'il est fini. Cette démarche est parfois appropriée mais, dans le cadre du projet, vous renverserez le processus pour vous appuyer sur la méthode dite de "développement piloté par les tests".

Pour chaque incrément, avant même d'écrire la première ligne de code, vous commencerez donc par écrire un jeu de tests, c'est à dire un ensemble de scripts `Scheme` **.scm** (typiquement une cinquantaine...) qui couvrent toutes les fonctionnalités attendues pour l'incrément et, pour chacun d'eux, un fichier **.res** contenant les sorties attendues (les résultats que devra produire votre programme pour le script de test considéré). Ce n'est qu'après avoir écrit votre jeu de tests que vous réfléchirez à la structure de votre programme et que vous le développerez.

Au début de l'incrément, aucun des tests du jeu de tests ne doit "passer". A l'issue de l'incrément, tous les tests que vous avez écrits doivent "passer" : votre programme doit produire le résultat attendu pour chacun des tests.

Voici quelques-uns des avantages du *développement piloté par les tests*.

— Cette méthode garantit que le programme sera accompagné d'un jeu de tests - alors que, trop souvent, les développeurs "oublient" de tester leur programme.

— Écrire les jeux de tests est un très bon moyen pour assimiler les spécifications du programme et s'assurer qu'on les a comprises. Le jeu de tests doit faire apparaître les situations simples, mais aussi faire ressortir les cas particuliers et plus complexes, ainsi que les cas où le programme doit détecter une erreur. Ce faisant, le développement lui même peut être guidé par une vue d'ensemble de ce que doit faire le programme, dans laquelle, dès le début, on envisage non seulement les cas simples, mais aussi les situations plus délicates.

— L'existence d'un jeu de tests permet de très vite détecter les *problèmes de régression*, c'est-à-dire les situations où, du fait de modifications introduites dans le code, une partie du programme qui fonctionnait jusqu'ici se met à poser problème. Or, la méthode incrémentale que vous adopterez tend à ce que le code soit souvent modifié pour tenir compte des nécessités d'un nouvel incrément... et donc à ce que des problèmes de régression surgissent.

Plus concrètement, vous vous astreindrez à faire tourner votre programme très souvent - le plus souvent possible, en fait - sur l'ensemble de vos tests ; cela vous permettra de détecter rapidement les problèmes et de vous assurer que votre programme, petit à petit, se construit dans la bonne direction.

Automatisation des tests ; script de test

Lancer tous les tests d'un jeu de test un à un est fastidieux. Il est donc fort utile d'avoir un environnement de test, capable de lancer tous les tests les uns après les autres et d'indiquer ceux qui sont en échec.

Nous vous fournissons pour ce projet un script (un programme écrit en Shell, dans le langage du Terminal), nommé `simpleUnitTest.sh`. `simpleUnitTest.sh` automatise le lancement des tests, et pour chacun d'eux vérifie que le programme testé retourne le bon code erreur et que la sortie standard du programme est bien conforme à ce qu'elle doit être. Ce script et son utilisation sont présentés dans la section "ce dont vous disposez pour commencer".

A propos de l'écriture des jeux de tests

L'écriture d'un jeu de tests pertinent - c'est-à-dire à même de raisonnablement "prouver" que votre logiciel fait ce qu'il doit faire - n'est pas chose évidente.

Pour vous mettre en jambe, nous vous donnons un jeu de tests déjà bien avancé pour le premier incrément. Il vous revient de le lire et d'en comprendre l'esprit.

Pour les incréments suivants, voici quelques conseils pour l'écriture des jeux de tests.

1. un bon jeu de tests comprend nécessairement de nombreux tests - par exemple une centaine pour chaque incrément.
2. commencer par écrire des scripts `Scheme` très courts et très simples. Par exemple, pour le troisième incrément relatif aux primitives, le script

   ```
   ( + 5 -4 )
   ```

   doit produire la sortie "1". La simplicité de ces scripts facilitera la recherche du problème si un test échoue.
3. penser également à écrire des scripts longs et/ou complexes : les problèmes peuvent surgir du fait d'instructions très longues ou d'enchaînements d'instructions !
4. se remuer les méninges face aux spécifications ; essayer d'imaginer les cas qui risquent de poser problème à votre programme ; pour chacun d'eux, écrire un test !

   Par exemple, pour le premier incrément consacré au *parsing*, le jeu de tests fourni joue autour des espaces : espaces multiples entre les mots, espaces constitués de tabulations ou de retour chariot, absence d'espace entre deux parenthèses ouvrantes ou fermantes, etc.
5. **important** : un jeu de tests se doit de tester ce qui doit marcher... mais aussi ce qui doit échouer et la façon dont votre programme gère les erreurs ! Pour ce faire, un jeu de tests doit *aussi* comprendre des scripts de tests qui comprennent des erreurs.

   Dans le cadre du projet, c'est d'autant plus important que l'interpréteur `Scheme` travaille sur des commandes saisies par un être humain, qui est par nature sujet à des erreurs de rédaction (absence de parenthèses fermantes, etc.).

   Par exemple, pour le second incrément consacré au début de l'évaluation, un script de test pourrait être :

   ```
   ( define 8 9)
   ```

   Ce script provoque une erreur et doit arrêter l'interpréteur : "8" n'est pas un nom de variable valide !

   Notez que le script `simpleUnitTest.sh` que nous vous fournissons pour lancer les tests est capable de détecter qu'un script censé arrêter l'interpréteur provoque bien une erreur.

Organisation interne d'un incrément

On a déjà indiqué que la réalisation de chacun des quatre incréments commencera par l'écriture du jeu de tests, et devra se terminer par un test du programme dans lequel tous les tests du jeu de tests "passent". Mais comment s'organiser à l'intérieur de l'incrément ? Voici quelques conseils...

1. réfléchir à la structure de votre programme. Définir en commun les principales structures de données et les signatures et "contrats" (ou "rôle") des principales fonctions.
2. Autant que faire se peut, essayer d'organiser l'incrément... de façon incrémentale ! La notion de développement incrémental peut en effet s'appliquer récursivement : à l'intérieur d'un incrément, il est souhaitable de définir des étapes ou "sous-incréments" qui vous assurent que votre programme est "partiellement fonctionnel" le plus souvent possible.

3. Réfléchir à la répartition du travail. En particulier, éviter que la répartition ne bloque l'un d'entre vous si l'autre prend du retard.
4. Compiler très régulièrement. Il est particulièrement fâcheux de coder pendant plusieurs heures pour ne s'apercevoir qu'à la fin que la compilation génère des centaines d'erreurs de syntaxe. Un développeur expérimenté fait tourner le compilateur (presque) tout le temps en tâche de fond !
5. Exécuter souvent le jeu de tests, afin de mesurer l'avancement et de faire sortir le plus tôt possible les éventuelles erreurs.

Ce dont vous disposez pour commencer

Le premier élément dont vous disposez est bien sûr ce sujet, qu'il vous appartient de lire, de relire, d'annoter et de comprendre. Un certain nombre d'autres éléments vous sont fournis.

Le canevas de programme

Le canevas de programme qui vous est fourni sait faire trois choses :

1. Il analyse tout d'abord les paramètres passés au programme en ligne de commande pour déterminer le flux d'entrée qui sera utilisé - l'endroit où les commandes `Scheme` seront lues. Ceci est effectué dans la fonction principale main() du fichier `repl.c`.

   Si le nom d'un fichier script, c'est-à-dire un fichier texte contenant du code `Scheme` est passé en paramètre du programme, le programme ouvre alors ce fichier, positionne le flux d'entrée au début du fichier, traite les commandes une à une, puis se termine. C'est ce mode qui est utilisé pour exécuter les tests.

   Sinon le programme est lancé en mode interactif : le flux d'entrée utilisé est l'entrée standard `stdin`, c'est-à-dire le clavier. Un prompt invite l'utilisateur à taper des commandes au clavier - comme dans un Terminal, si ce n'est qu'ici ce sont des commandes en langage `Scheme` qui sont bien sûr attendues.
2. le canevas de programme comprend une fonction `sfs_get_sexpr()` - codée dans le fichier `read.c` et appelée depuis le main() - dont le rôle est de récupérer la prochaine S-expression "bien formée" dans le flux d'entrée, et de la stocker dans une chaine de caractère.

   Le contrat de cette fonction est :

```
uint  sfs_get_sexpr( char *input, FILE *fp );
```

   `sfs_get_sexpr()` lit dans le flux `fp` la prochaine S-expression puis stocke cette S-expression dans la chaine de caractère `input`. Détail : La fonction commence par lire dans `input` une ligne, puis compte le nombre de parenthèses ouvrantes et fermantes sur la ligne. S'il y a plus de parenthèse fermantes qu'ouvrantes, la fonction retourne `FALSE`. S'il y a plus de parenthèse ouvrantes que fermantes, la fonction ajoute la ligne suivante dans `input`, et continue le compte des parenthèses. Si le nombre de parenthèse ouvrantes et fermantes est identique, la fonction retourne `TRUE`. Attention : en conséquence, les trois entrées suivantes :

```
a b c
(qqchose) (autrechose)
(qqchose) 78
```

seront considérées comme des S-sxpressions bien formées et la fonction retournera `TRUE` ... alors qu'en fait ces expressions sont invalides !

PRé-CONDITIONS : le flux `fp` doit être préalablement ouvert en lecture et la chaîne `input` doit être préalablement allouée en mémoire.

VALEUR DE RETOUR : Si une S-expression est trouvée, la fonction retourne `TRUE`. Sinon, elle retourne `FALSE`.

3. Une fois la S-expression acquise, le programme lance la boucle d'interprétation en appelant successivement, depuis le main(), les fonctions `sfs_read()` (dans le fichier `read.c`), `sfs_eval()` (dans `eval.c`) puis `sfs_print()` (dans `print.c`).

   Toutefois, dans le code fourni, ces trois fonctions ne font rien du tout : votre rôle consiste précisément à les écrire au fur et à mesure des incréments.

Le canevas de programme définit également le début de la structure `object`, dans le fichier `object.h`, telle qu'elle a été introduite dans la premièrre partie du sujet.

Enfin, quelques fonctions et macros utilitaires vous sont fournies. En particulier, une macro `DEBUG()`, définie dans le fichier `notify.h`, vous permettra d'afficher des messages de *debug*. Cette macro s'utilise comme `printf()`.

Notez que votre interpréteur ne doit écrire dans la sortie standard `stdout` (au moyen de `printf()` ou `fprintf(stout, ...)`) que les résultats attendus et rien d'autre. Tous les messages de *debug*, ou d'erreur, doivent donc systématiquement être écrits dans le flux d'erreur standard `stderr` (au moyen de la macro `DEBUG()`, ou de `fprintf( stderr, ...)` si vous préférez). Cela est nécessaire pour permettre d'automatiser les tests en comparant ce que sort votre interpréteur dans la sortie standard avec ce qui est attendu.

Le Makefile

Un `Makefile` vous est fourni pour compiler le canevas de programme. Il vous appartient de faire évoluer ce `Makefile` au fur et à mesure du développement, notamment dès que vous introduisez un nouveau fichier source .c.

Pour compiler le programme `scheme` en mode *debug*, afin de pouvoir le faire tourner dans un *debugger* (tel que `ddd` ou `gdb`) et pouvoir afficher les messages de *debug* (cf. `notify.h`), taper dans un terminal :

```
make debug
```

Pour le compiler en mode *release* (optimisé), taper :

```
make release
```

Pour nettoyer toute la compilation (supprimer tous les fichiers objets intermédiaires), taper :

```
make clean
```

Pour faire une archive de tout votre code (par exemple pour l'envoyer aux enseignants, ou pour faire des sauvegardes de votre travail), taper :

```
make tarball
```

Le script de test

L'archive fournie comprend un script Shell `simpleUnitTest.sh` qui permet d'exécuter un jeu de tests et de récupérer à la fin la liste des tests réussis et des tests en échec.

Ce script figure dans le répertoire `testing/` de l'archive.

Le fichier `README.txt` qui accompagne ce script décrit son mode d'emploi ainsi que la marche à suivre pour écrire un nouveau jeu de tests.

Notez que, pour que `simpleUnitTest.sh` fonctionne, chaque test doit être constitué :

— d'un fichier texte **.scm** comprenant le code `Scheme` à tester. C'est ce script `Scheme` que votre interpréteur exécutera. Attention : ce fichier doit commencer par deux lignes de commentaires `Scheme` particuliers, destinées à `simpleUnitTest.sh` (explications dans le fichier `README.txt`).

— d'un fichier texte **.res** dans lequel vous précisez le résultat qu'est censé fournir (afficher) votre programme sur sa sortie standard (`stdout`). `simpleUnitTest.sh` comparera le contenu de ce fichier **.res** avec la sortie standard `stdout` générée par votre programme.

Le jeu de tests du canevas de programme

Ce premier jeu de tests permet de tester le bon fonctionnement du canevas de programme qui vous est fourni. Il figure dans le répertoire `scheme/tests_step0`.

Pour exécuter tous les tests à l'aide du script de test :

— aller dans le répertoire `scheme/` et compiler le programme avec `make debug`

— exécutez tous les tests avec la commande :

```
../testing/simpleUnitTest.sh -e ./scheme tests_step0/*.scm
```

Dans cette dernière commande :

— "`../testing/simpleUnitTest.sh`" est le chemin vers le script de test

— "`-e ./scheme`" indique le chemin vers le programme à tester

— "`tests_step0/*.scm`" liste l'ensemble des scripts `Scheme` qui constitue le jeu de test.

Vous constaterez que tous les tests du répertoire `tests_step0` passent sans échec [1] ; c'est normal, ils ne testent que la fonction `sfs_get_sexpr()` qui existe déjà !

Le jeu de tests du premier incrément

Dans le répertoire `scheme/tests_step1`, vous trouverez une série de tests pour le premier incrément du projet. Vous remarquerez que deux jeux de tests sont fournis : des tests basiques (répertoires `simple`) et des tests plus complets (répertoire `evolved`).

Pour lancer tous ces tests, aller dans le répertoire `scheme/` et taper :

```
../testing/simpleUnitTest.sh -e ./scheme tests_step1/simple/*.scm
../testing/simpleUnitTest.sh -e ./scheme tests_step1/evolved/*.scm
```

Vous constaterez que tous les tests sont pour le moment en échec... C'est normal, puisque le code correspondant au premier incrément du projet n'existe pas encore ! A l'issue de l'incrément, tous les tests devront passer sans erreur.

Notez que les tests fournis pour le premier incrément constituent une bonne base de départ, mais que vous pouvez la compléter si il vous apparait que certaines des fonctionnalités attendues ne sont pas (ou pas assez) testées...

Le site internet du projet

Vous trouverez l'archive contenant le squelette de programme, le Makefile, le script de test et les premiers jeux de tests sur le site internet du projet. Des informations complémentaires pourront vous être données sur ce site au fur et à mesure du projet.

1. si un test est en échec, c'est qu'une (malencontreuse ?) erreur s'est glissée dans le canevas de programme fourni. Il vous appartient, alors, de la corriger...

Conseils pour la rédaction du code

Variables globales

Les variables globales sont le Mal[2]. Souvent, il n'est pas très difficile de s'en passer : il faut juste rajouter, la plupart du temps, un argument supplémentaire dans les appels de fonctions. Lorsqu'on a pléthore de variables qu'on a envie de déclarer globalement, on les regroupe généralement dans une structure qui encode alors l'état général de notre programme, et on passe cette structure en paramètre supplémentaire aux fonctions ayant à connaître et/ou modifier l'état global du programme. C'est souvent comme cela que font les gens civilisés.

Dans notre cas, nous aurons des symboles d'intérêt global en pagaille (la liste vide, les deux booléens, l'environnement meta, la table des symboles, *etc*). *Exceptionnellement*, et uniquement parce que ce n'est pas vraiment le plus important dans ce projet, nous vous autorisons à déclarer des variables globales.

Considérons par exemple l'objet global désignant la liste vide. Supposons que la liste vide soit déclarée dans le fichier `main.c` (cf. Fig. 3.1).

Maintenant, lorsqu'une fonction dans un autre fichier a besoin de renvoyer la liste vide, il faudra penser à déclarer l'objet liste vide avec le qualifieur `extern` (puisqu'il est déclaré dans un autre fichier). Par exemple, si l'évaluation échoue et qu'on renvoit la liste vide pour le signaler, alors on aura quelque-chose ressemblant au bout de code contenu dans le fichier `eval.c` (Fig. 3.2).

Au sujet des objets, des formes et des primitives

Vous gagnerez à définir des fonctions de création pour chaque type d'objet (cf. Fig. 3.1 pour la liste vide), ainsi que des fonctions permettant de tester le type d'un objet, et d'en afficher le contenu.

De la même manière, vous gagnerez à définir des fonctions permettant de tester si une S-expression correspond à telle ou telle forme.

Concernant les primitives (initialisées au lancement de l'interpréteur), vous aurez intérêt à définir une fonction servant tout à la fois à déclarer leur symbole, et à leur associer leur fonction. Une telle fonction aura donc comme prototype :

```
int declare_primitive( char *symbole, object (*fonction)(object ) );
```

Lorsque vous aurez défini votre primitive pour `cons`, par exemple :

```
object cons_primitive( object sexpr_cdr );,
```

alors, dans l'initialisation de l'interpréteur, vous écrirez :

declare_primitive(" cons", cons_primitive) ; .

Du code lisible...

... pour ce qu'on fait !

Pour les formes, mais pas uniquement, vous aurez intérêt à définir des fonctions accessoires permettant de rendre votre code lisible. Par exemple, pour la forme `if`, plutôt que d'aller chercher directement le `cadr` pour obtenir le prédicat, vous définirez une fonction dont le nom sera `predicat`, et qui ira effectivement chercher le `cadr` de la S-expression de la forme `if`. Vous agirez de même pour créer les fonctions `consequence` et `alternative`. Vous procéderez de même chaque fois que vous le pourrez.

2. Quelle fonction les modifie quand ? *etc.*

```
#include <stdlib.h>  /* exit, EXIT_SUCCESS */
#include <stdio.h>   /* FILE */

#include "scheme.h"  /* Fonctions et objets du projet Scheme */

/* Globals */
object empty_list;
object meta_envt;

/* Init */
void init() {

  empty_list = make_empty_list();
  meta_envt = make_global_envt();

}

/* REPL */
int main ( int argc, char *argv[] ) {
  object user_entry;
  FILE *stream = stdin;

  init();

  while( 1 ) {
    user_entry = read( stream );

    print( eval( user_entry, meta_envt ) );
  }
  exit( EXIT_SUCCESS );
}
```

Figure 3.1 – Exemple de squelette de code pour les méthodes `init()` et `main()`. Le code que nous fournissons s'inspire et étend ces fonctionnalités.

D'une manière générale, vous pouvez vous débrouiller pour n'avoir *quasiment aucun* appel aux fonctions `car` ou `cdr` dans votre code, excepté dans des fonctions accessoires.

... pour ce qu'on manipule !

Vous remarquerez également que vous allez toujours manipuler, en C, le même type d'objet (celui de la Fig. 1.6). Ce type d'objet permet de tout représenter, les nombres comme les S-expressions. C'est la situation idéale pour obtenir un code illisible. Nous vous proposons de spécifier dans le nom de la variable en C le type effectif que vous manipulerez dans la fonction. Par exemple, pour afficher

```
object eval( object sexpr ) {

  extern object empty_list;

  /* evaluation... */

  /*
  Si l'evaluation reussit, on renvoie ci-dessus
  le resultat de l'evaluation. Par defaut, en
  cas d'echec, on renvoie la liste vide.
  */

  fprintf( stderr, ''Eval failed!\n'' );
  return empty_list;
}
```

Figure 3.2 – Extrait de code pour le fichier `eval.c`

un nombre, vous ne définirez pas la fonction
`void affnb( object o );`
mais plutôt :
`void afficher_nombre( object nombre );`.

Gestion mémoire

Une remarque capitale au sujet de la gestion mémoire est que, jusqu'à présent, on n'a fait que créer des objets. On ne les libère jamais ! Votre code ne contient que des `malloc(3)` et aucun `free(3)`... Voilà qui semble contredire tout ce que nous vous avons raconté en première année[3].

Ce problème n'en est pas vraiment un, et découle de la nature même d'un interpréteur : il n'y a pas de moyen simple de déterminer quand un objet ne sert plus et donc quand la mémoire qu'il occupe peut être libérée. La seule solution est alors d'utiliser un ramasse-miettes (*garbage collector* en anglais).

Il existe en pratique deux solutions pour inclure un ramasse-miettes dans votre projet :

1. soit vous le réalisez à la main ;
2. soit vous en utilisez un déjà tout fait.

Si vous choisissez la seconde solution, alors le meilleur choix est probablement d'utiliser le ramasse-miettes Boehm-Demers-Weiser [2]. Reportez-vous aux exemples sur la page citée dans la bibliographie pour une utilisation de base, amplement suffisante pour ce projet.

Si, par contre, vous souhaitez développer vous-mêmes un ramasse-miette, vous détaillerez la solution que vous aurez conçue. Une étape obligée sera de rajouter un champ `unsigned int refcount` dans la structure décrivant les objets. Ce champ sera incrémenté à chaque fois qu'un objet sera référencé quelque-part (par exemple, dans la Fig.1.1, l'objet symbole `i` aurait un champ `refcount`

3. à chaque `malloc(3)` son `free(3)` !

valant 2). On sait qu'on pourra libérer la mémoire d'un objet lorsque son `refcount` vaudra zéro (l'objet n'est plus référencé nulle part). Vous vous demanderez en particulier quand incrémenter et décrémenter le champ `refcount` des objets, et quand libérer effectivement la mémoire.

Chapitre 4

Travail à réaliser : étapes et barème

Notes :
— Pour chaque étape, vous placerez le code correspondant dans un répertoire distinct ;
— Les prototypes des fonctions que nous vous donnons sont des prototypes minimaux, que vous pouvez étendre au besoin.

Étape 1 : Analyse lexicale et affichage

But

Deux des fonctionnalités peut-être les plus délicates à implanter sont sans doute l'analyse lexicale d'une commande pour construire "l'arbre syntaxique" correspondant ("parsing") et l'affichage de l'arbre qui encode le résultat de cette commande. Il est donc intéressant de commencer par réaliser une fonction de lecture (qui construit "l'arbre syntaxique" correspondant à la commande) et une fonction d'écriture (qui affiche un arbre donné à l'écran). C'est le but de ce premier incrément.

À la fin de cette étape, vous devrez avoir validé l'analyse lexicale. Les deux fonctions principales que vous devrez avoir implantées sont : `read` et `print`.

On ne s'occupe pas de l'évaluation à cette étape. Pour cela, vous utiliserez la fonction suivante pour l'évaluation :

```
object eval( object expression ) { return expression; }
```

Votre boucle REPL se comportera alors bêtement comme un automate ne sachant que répéter en sortie ce qui a été aura lu sur l'entrée - en supprimant toutefois les espaces inutiles, c'est-à-dire en produisant une sortie "propre". Mais vous aurez ainsi validé la création en mémoire d'arbres syntaxiques de S-expressions correctes.

Ce n'est que dans l'incrément suivant que vous pourrez vous consacrer à l'étape d'évaluation pour faire enfin "vivre" votre interpréteur.

Jeu de tests

Un jeu de tests pour ce premier incrément vous est donné. Votre rôle se limite donc à le lire, le comprendre et le cas échéant le critiquer et le compléter.

Au début de l'incrément, aucun test du jeu n'est censé passer puisque le code correspondant n'existe pas encore dans votre programme. A la fin de l'incrément, tous les tests doivent passer.

Marche à suivre

Définition du type de donnée générique, initialisation et instanciation de nœuds

Vous vous limiterez pour l'instant à la structure de la Fig. 1.6, qui incidemment vous est déjà fournie en grande partie dans le fichier `object.h` du canevas de programme.

Vous définirez des variables globales pour stocker les objets suivants : `empty_list`, `true` et `false`. Vous écrirez les fonctions nécessaires à leur création, et vous écrirez une fonction d'initialisation de l'interpréteur qui appelle ces fonctions, de telle sorte qu'il existe, dès le lancement du programme, une instance de `empty_list`, `true` et `false`.

Ces fonctions sont :

— `object make_empty_list( void )`

— `object make_boolean( unsigned int )`

Vous écrirez ensuite les fonctions permettant de créer des symboles, des entiers, etc. (*i.e.* des fonctions pour tous les types de base), dont les prototypes sont :

— `object make_symbol( char * )`

— `object make_integer( int )`

— *etc.*

La gestion des nombres en virgule flottante ("réels") et des nombres complexes sont des bonus pour ce projet. Si vous décidez de les prendre en compte, votre analyseur lexical devra pouvoir les reconnaître et devra disposer des fonctions de construction appropriées. Notez que la structure `object` du canevas de code qui vous est fourni a été préparée pour cela puisqu'elle définit une structure `num` et le type d'`object` `SFS_NUMBER` (un nombre pouvant être un entier, un réel ou un complexe) et non pas uniquement un type `SFS_INTEGER`.

Fonction `read()`

La fonction `read` est le point d'entrée de l'analyseur lexical ADR. Vous utiliserez le prototype suivant :

```
object read( char * );
```

Cette fonction doit construire l'arbre syntaxique correspondant à une S-expression lue dans le flux d'entrée. Comme expliqué précédemment, cette fonction doit d'abord détecter si on a affaire à une paire (ou liste) ou à un atome, et appeler `read_pair()` ou `read_atom()` en conséquence. Pour faciliter votre travail, vous pourrez commencer par considérer que votre analyseur ne sait lire que des atomes et vous contenter d'appeler `read_atom()`. Vous pourrez ensuite compléter cette fonction lorsque vous écrirez la fonction `read_pair()`.

Fonction `print()`

La fonction `print` est le point d'entrée de l'affichage du résultat de votre interpréteur. Son rôle est de convertir un arbre syntaxique en chaîne de caractères et de l'afficher dans la sortie standard (avec `printf()`).

Vous utiliserez le prototype suivant :

```
void print( object );
```

En fonction du type de l'`object` passé en paramètre (paire ou atome), la fonction devra appeler `print_pair()` ou `print_atom()`. Comme pour `read()`, vous pourrez considérer dans un premier temps que votre interpréteur ne sait que manipuler des atomes et vous contenter d'appeler `print_atom()`, puis compléter la fonction lorsque vous vous occuperez de `print_pair()`.

Écriture de la fonction `read_atom()`

Vous utiliserez le prototype suivant :

```
object read_atom( char * );
```

Cette fonction doit être capable de reconnaître n'importe quel type d'atome, et de renvoyer un pointeur sur un `object` qui contient la valeur de l'atome lu.

Attention au comportement de cette fonction ! Lorsque vous lirez la liste vide ou les booléens `#t` ou `#f`, vous devrez renvoyer l'adresse de l'objet que vous aurez créé lors de l'initialisation de votre interpréteur et non pas créer un nouvel objet.

Écriture de la fonction `print_atom()`

Vous utiliserez le prototype suivant :
`void print_atom( object );`
Cette fonction vous permettra de valider la précédente en affichant correctement un atome, en particulier son type (sous forme d'une chaîne de caractères) et sa valeur.

Écriture de la fonction `read_pair()`

Vous utiliserez le prototype suivant :
`object read_pair( char * );`
Cette fonction doit être capable de lire n'importe quelle liste (avec, ou constituée de, la liste vide) et appeler la fonction `read` pour lire les éléments contenus dans les paires. Vous remarquerez que les fonctions `read` et `read_pair` sont mutuellement récursives (cf. mécanisme d'un ADR).

Écriture de la fonction `print_pair()`

Vous utiliserez le prototype suivant :
`void print_pair( object );`
Cette fonction doit permettre d'afficher n'importe quelle liste (avec, ou constituée de, la liste vide) et appeler la fonction `print()` pour afficher les éléments contenus dans les paires. Bien évidemment, par symétrie, les fonctions `print()` et `print_pair()` seront elles aussi mutuellement récursives.

Gestion des symboles

Pour ce premier incrément, vous n'implanterez pas encore de table des symboles. Lorsqu'un symbole (un nom de variable) existe dans le code `Scheme` à analyser, vous vous contenterez d'instancier un objet de type `SFS_SYMBOL` qui comprend le nom de ce symbole en appelant la fonction `make_symbol()` évoquée plus haut.

Étape 2 : Évaluation, formes et environnements

But

C'est à partir de cette étape que notre interpréteur commencera à vivre car nous allons nous occuper de la fonction d'évaluation. À la fin de cette étape, vous serez en mesure de définir des variables dans l'environnement et d'utiliser quelques formes, comme par exemple : `quote`, `define`, `set!` et `if`.

Il est important de bien comprendre le mécanisme par lequel nous allons implanter les formes. Une forme est d'abord identifiée par son nom, c'est donc avant tout un symbole qu'il convient de créer dès l'initialisation.

Lorsque nous évaluerons une expression, nous regarderons d'abord si son `car` est un symbole correspondant à une forme. Si c'est la cas, nous procédons à l'évaluation particulière de cette forme.

Jeu de tests

Votre première tâche pour cet incrément est de définir un jeu de tests couvrant les fonctionnalités attendues. Vous vous référerez aux conseils fournis dans la section 3 du sujet. Au début de l'incrément,

aucun test de votre jeu ne passera, puisque le code correspondant n'existe pas encore dans votre programme. A la fin de l'incrément, tous les tests doivent passer.

Marche à suivre

La forme `quote`

Pour commencer dans la continuité de la première étape, vous vous interéresserez à la forme `quote`.

Votre interpréteur doit pouvoir reconnaître de manière équivalente `(quote a)` et `'a`. En réalité, lorsque l'utilisateur rentre `'a`, vous devez convertir l'expression en `(quote a)`.

Au moment de l'évaluation de l'expression, vous détecterez s'il s'agit d'une forme `quote`. Dans l'affirmative, vous renverrez naturellement le `cdr` de l'expression comme résultat de l'évaluation.

Environnements

Vous implanterez une table des symboles sous forme de variable globale, telle que décrite dans la Fig. 1.1.

Une table de symboles à un seul niveau peut être encodée au moyen d'une simple liste chaînée, dont chaque élément contient à la fois le nom du symbole et sa valeur.

Notez que vous disposez déjà dans votre interpréteur Scheme d'un moyen permettant de manipuler des listes chainées : les listes `Scheme` ! Au lieu de redéfinir un nouveau type pour les listes, vous vous astreindrez à utiliser une liste `Scheme` pour stocker votre table de symboles.

Par ailleurs, vous prévoirez dès maintenant de pouvoir imbriquer les environnements comme décrit sur la Fig. 1.1. Il vous appartient de définir la structure de la liste `Scheme` qui permettra de stocker plusieurs tables de symboles dans des environnements imbriqués...

Vous écrirez ensuite les quelques fonctions `C` facilitant la manipulation de vos environnements imbriqués :

- Fonctions de gestion des environnements et des symboles.
- Fonction permettant d'ajouter une variable dans l'environnement courant. Lorsque vous lirez un symbole (par exemple passé en paramètre à la forme `define`), vous regarderez s'il existe déjà dans la table des symboles. Si oui, vous renverrez son adresse. Si non, vous le créerez, l'ajouterez dans la table des symboles, et renverrez son adresse. La fonction `make_symbol` introduite dans le premier incrément sera modifiée pour inclure ce fonctionnement.
- Fonction permettant de rechercher la valeur d'une variable dans tous les environnements situés sous l'environnement courant (inclus). Si une variable n'est pas définie, vous renverrez la liste vide en affichant un message d'erreur (par exemple, le fameux `Unbound variable`!)
- ...

Les formes `define` et `set!`

Vous implanterez la forme `define` de telle manière que l'on puisse rajouter une variable dans l'environnement courant. Par convention, le résultat de l'évaluation de la forme `define` est le nom de la variable qui vient d'être définie.

Vous implanterez la forme `set!` permettant de modifier la valeur d'une variable déjà définie avec `define`. Vous rechercherez la variable dans tous les environnements situés sous l'environnement courant (inclus), en vous arrêtant sur la première occurence que vous trouverez, et vous renverrez sa valeur comme résultat de l'évaluation de la forme `set!`. Vous renverrez la liste vide et un message d'erreur si la variable n'est pas définie.

La forme `if`

Vous implanterez la forme `if` telle que décrite dans la Sec. 1.3.7. Votre interpréteur sera alors Turing-complet. Pour cela, vous remarquerez qu'il faut d'abord évaluer le prédicat, puis, en fonction, évaluer la conséquence ou l'alternative. Nous avons deux évaluations à mener. Une solution pourrait être de rappeler la fonction `eval`, mais comme on s'attend à utiliser principalement la récursivité en `Scheme`, on implantera cette double évaluation avec un `goto`[1]. En effet, nous évitons ainsi toute une foultitude d'appels à `eval` et nous évitons de saturer la pile. Nous réutiliserons ultérieurement cette astuce pour d'autres formes.

Les formes `and` et `or`

Vous terminerez cet incrément en douceur avec l'implantation des formes `and` et `or`. Ces formes ont ceci de particulier que, tout comme en C, elles ont la propriété de court-circuiter : dès qu'un opérande de `or` est vrai, ou qu'un des opérande de `and` est faux, l'évaluation du prédicat est arrêtée puisqu'on en connait le résultat dès ce moment.

Étape 3 : Primitive galore

But

La manière dont nous implanterons les primitives est différente de celle dont nous avons implanté les formes précédentes. Une primitive est aussi un symbole qu'il faut déclarer lors de l'initialisation. Cependant, il suffit juste de reconnaître ce symbole et de lancer la fonction associée plutôt que, comme pour les formes, modifier la fonction d'évaluation. Vous aurez donc besoin de modifier la structure de type générique comme dans la Fig. 1.12 pour rajouter le type `primitive`.

Jeu de tests

Votre première tâche pour cet incrément est (bien sûr...) de définir un jeu de tests couvrant les fonctionnalités attendues.

Marche à suivre

Addition variadique

Afin de vous faire la main avec les primitives, vous commencerez par implanter l'addition de nombres entiers. En `Scheme`, l'addition renvoie le total de tous les arguments de type nombre entier de l'expression. Étonnamment, vous choisirez le symbole + pour cette primitive (voir les autres primitives d'arithmétique entière ci-dessous).

Moult autres primitives

Vous implanterez les primitives suivantes :
— Prédicats :
  — `null?`,

1. Eh oui, les `goto` existent en C ! L'utilisation abusive de `goto` nuit à la santé mentale du programmeur, mais là, il s'agit d'un cas de force majeure.

— `boolean?`,
— `symbol?`,
— `integer?`,
— `char?`,
— `string?`,
— `pair?`.
— ...

Un prédicat renverra vrai si tous les arguments sont du type testés, et faux sinon. Le prédicat `null?` teste la liste vide ;

— Conversions de types :
— `char->integer`,
— `integer->char`,
— `number->string`,
— `string->number`,
— `symbol->string`,
— `string->symbol` ;
— ...

— arithmétique entière :
— `-`,
— `*`,
— `quotient`,
— `remainder`,
— `=`,
— `<`,
— `>`.

Vous ferez en sorte, comme pour l'addition, de gérer un nombre quelconque d'arguments ;

— manipulation de listes :
— `cons`,
— `car`,
— `cdr`,
— `set-car!`,
— `set-cdr!`,
— `list` - cette primitive renvoie la liste formée des arguments passés en paramètre .

— égalité polymorphique : `eq?`. Cette primitive teste si les arguments désignent la même entité.

Note sur l'arithmétique flottante et complexe : la prise en compte des nombres en virgule flottante et des complexes est un bonus. Si vous considérez ces types de nombres, il faudra en tenir compte dans les opérations arithmétiques.

Étape 4 : Agrégats et forme `begin`, λ-calcul et formes `lambda` et `let`

But

Cette étape est probablement la plus intéressante de ce projet. Il s'agit dans une première partie de compléter la forme `define` afin de pouvoir lui faire accepter des fonctions. Pour cela, il faudra d'abord décrire à notre interpréteur ce qu'est un agrégat (on rappelle qu'il s'agit d'une fonction *sans son nom*). Une fois les agrégats décrits et implantés, on pourra rajouter la forme `begin`, qui exécute un *bloc d'intructions* – il s'agit donc de quelque-chose de très proche d'un agrégat, et on utilisera

naturellement un agrégat pour cela.

La seconde partie consiste à introduire le *lambda*-calcul dans notre interpréteur. Il s'agit principalement de faire s'exécuter un agrégat dans un environnement créé spécialement pour l'occasion. Une fois la forme `lambda` implantée, vous pourrez vous intéresser à la forme `let`, qui se code en la transformant au sein de l'interpréteur (dans le dos de l'utilisateur) en une forme `lambda`.

Accessoirement, vous rajouterez le prédicat `procedure?` qui renverra `#t` pour les primitives et les agrégats. De même, lorsque l'utilisateur entrera le nom d'une fonction ou une forme `lambda`, l'interpréteur renverra `#<procedure>`.

Jeu de tests

Eh bien, devinez... Votre première tâche pour cet incrément est de définir un jeu de tests couvrant les fonctionnalités attendues.

Marche à suivre

Agrégats : `define` mis à jour et `begin`

Vous commencerez par modifier le type générique pour prendre en compte les agrégats (*compounds*). Vous coderez ensuite des fonctions permettant de créer et exécuter un agrégat (vous penserez, pour la sous-étape suivante, à réserver un paramètre supplémentaire pour spécifier l'environnement dans lequel exécuter l'agrégat). Vous lierez ensuite ces fonctions au traitement (lecture et évaluation) de la forme `define`.

Pour vous reposer, vous coderez ensuite la forme `begin`.

λ-calcul : formes `lambda` et `let`

Vous ferez en sorte de pouvoir exécuter un agrégat dans un environnement donné (spécifié en paramètre), cf. ci-dessus. Vous rajouterez la forme `lambda`, qui consiste à créer un environnement et à y exécuter le corps de la fonction anonyme (vous utiliserez donc un agrégat pour décrire une fonction `lambda`).

Pour terminer avec le λ-calcul, vous pourrez rajouter la forme `let` – elle se code en la transformant à la volée en une forme `lambda`.

Prédicat `procedure?` et évaluation

Bonus

Vous avez désormais en votre possession un interpréteur permettant de développer un programme en utilisant exclusivement le langage `Scheme`, et non plus le langage C.

A titre d'exemple, vous disposez de tous les outils de base pour rajouter le paradigme de programmation objet à votre interpréteur, mais en le codant en `Scheme` cette fois ! Certes, ce serait un peu pénible, mais faisable (vous vous demanderez comment faire une fois votre cours de POO terminé...).

Quoi qu'il en soit, vous pourrez constater que le "style de programmation" associé au paradigme de programmation fonctionnelle est bien différent de celui auquel vous êtes habitué avec le C : avec le recours systématique aux listes et l'utilisation de la récursivité, le `Scheme` a un très fort pouvoir expressif.

Complétude de l'arithmétique

Vous pourrez, pour commencer, compléter la gestion de l'arithmétique : nombres réels et complexes si ce n'est déjà fait, mais aussi, pourquoi pas, nombres rationnels. Remarquez qu'ajouter un nouveau type de nombre - ainsi que les nouvelles opérations associées - nécessite de modifier une bonne partie du code C (structure *object* ou champs *number* de cette structure, *parsing* pour lire ce nouveau type, primitives...), mais que cela devrait être très facile si votre code est bien organisé...

Entrés-sorties

Une extension importante consiste à ajouter de quoi gérer les entrées / sorties depuis et vers des fichiers. Doté d'entrées / sorties, votre interpréteur devient en effet capable de traiter tout type de données ! Les primitives concernées s'appellent :

- `load`,
- `read`,
- `read-char`,
- `peek-char`,
- `input-port?`,
- `open-input-file`,
- `close-input-file`,
- `eof-object?`,
- `write`,
- `write-char`,
- `output-port?`,
- `open-output-file`,
- `close-output-file`,
- `error`.

La Sec. 6.6 du R^{5}RS [3] détaille ces fonctions.

Forme `eval`

L'ajout de la forme `eval`, qui procède à l'évaluation d'une liste considérée comme une commande `Scheme` dans un environnement donné, est une autre extension importante. Avec `eval` et les entrées / sorties, il devient par exemple possible d'écrire un interpréteur `Scheme` ... en `Scheme` ! Ou, plus généralement, il devient possible de générer n'importe quel programme `Scheme` en `Scheme`, puis de l'exécuter...

Ramasse-miettes

Comme indiqué précedemment, nous ne nous sommes pas du tout préoccupés de la gestion de la mémoire pour le moment. Votre interpréteur ne sait qu'allouer de la mémoire, mais ne la libère jamais, même lorsqu'elle n'est plus utilisée ! Face à ce défaut important, vous pouvez penser à ajouter un ramasse-miette, en expliquant quelle stratégie vous aurez adoptée.

Librairie standard

Vous pourrez également créer une ébauche de librairie standard en `Scheme`, chargée dès le lancement de l'interpréteur, et incluant des fonctions permettant par exemple d'appliquer une fonction à

tous les éléments d'une liste, de renverser une liste (le premier est échangé avec le dernier, le second avec le pénultième, etc.), de calculer la longueur d'une liste, de concaténer deux listes, *etc*. Le tout écrit en `Scheme` désormais !

Annexe A

Exemple de programme `Scheme`

Vous trouverez ci dessous en Figure A.1, un exemple de programme `Scheme` composé d'une succession d'expressions exécutées les unes à la suite des autres. Le code est sensé être suffisamment commenté pour être compréhensible sans plus de description. Nous vous encourageons à exécuter celui-ci dans un interpréteur `Scheme` pour obtenir les résultats d'évaluation.

```
; le debut d'un commentaire est marque par un ";"
; (commande ARG1 ARG2 ...)
(+ 1 2 3 4) ; s'evalue a 10
(+ 1 2 3 4) ; commentaire
(define a 10) ; lie la variable 'a' a la valeur 10
a             ; evalue 'a' qui vaut 10
(* a 9)       ; examples de calcul
(* a (+ 4 5))

(quote something)    ; affiche le symbole something sans l'evaluer
'something           ; pareil...
(quote (* a (+ 4 5))); affiche l'expression (* a (+ 4 5)) sans l'evaluer

; nombres, caractere, chaine, booleens
5                ; 5 s'auto evalue a 5
#\a              ; de meme pour les atomes suivants
"chaine de car"
#t

; listes
(cons 1 2)       ; une paire
(list 1 2 3)     ; une liste (fini par un ())

(car (list 1 2 3)) ; renvoie la tete 1
(cdr (list 1 2 3)) ; renvoie la queue (2 3)

; conditionnelle
(and (= 4 3) (< 4 9)) ; evalue et renvoie #t

(define maVar 3)  ; binding de variable entre la valeur 3 et le symbole maVar

(if (= maVar 3) (quote "equals") (quote "not equal"))

; lambda
((lambda (x) (* 2 x)) 5)  ; definition et execution d'une fonction anonyme
                          ; renvoie 10, la fonction n'est pas conservee

(define power2 (lambda (x) (* x x))) ; declaration et definition de power2
                                     ; par binding de l'indentifiant power2
                                     ; et de la fonction anonyme

; definition de la fonction factorielle
(define factorial (lambda (n)
    (if (= n 0)
        1
        (* n (factorial (- n 1))))))

; utilisation de la fonction factorielle
(factorial 4)

(map (lambda (x) (+ x 1)) (list 2 3 4)) ; utilisation du mot-cle map
                                        ; la fonction lambda est appliquee
                                        ; a chaque element de la liste
                                        ; map retourne une liste contenant les
                                        ; nouvelles valeurs
```

Figure A.1 – Exemple de programme Scheme.

Bibliographie

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : Principles, Techniques and Tools*. Addison Wesley, 2006. [WWW].

[2] Hans Boehm, Alan Demers, and Mark Weiser. A Garbage Collector for C and C++. 1988. [WWW].

[3] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 Report on the Algorithmic Language Scheme. 1998. [WWW].

[4] Pierre Lescanne. Et si on commençait par les fonctions ! 2009. [WWW].

[5] John McCarthy. Recursive Functions of Symbolic Expressions and their Computation by Machine (part 1). *Communications of the ACM*, 3,4 :184–195, Apr. 1960. [PDF].