Sémantique de Scheme en $\mathbb{K}$

Table des matières

I. À propos de Scheme

Scheme est, parmi les dialectes de Lisp, celui considéré comme le plus épuré, conçu notamment pour la formation à la programmation, et qui a connu le plus d'enrichissements liés à des développements théoriques. Parmi ces enrichissement par rapport à d'autres dialectes de Lisp, on peut citer l'opérateur `call/cc` et les nombreuses variantes qui en ont été proposées, et le système de macros hygiéniques (qui a d'ailleurs fortement inspiré celui de Rust).

Il existe de nombreuses implémentations de Scheme, avec des différences importantes. Un effort d'uniformisation a été fait avec les versions successives du *Report on the Algorithmic Language Scheme*, appelées *Revised*^{n} *Reports on the Algorithmic Language Scheme* (pour la n-ième révi-

sion), nom abrégé en « R^{n}RS ». Les derniers en date sont le R^{5}RS (1998), le R^{6}RS (2007), et le R^{7}RS « small ». Le R^{6}RS s'est démarqué du R^{5}RS et de ses prédécesseurs en se montrant bien plus ambitieux : il définit un langage beaucoup plus gros que le R^{5}RS (ce qui est visible simplement d'après leurs nombres de pages respectifs), en standardisant plus de fonctionnalités. Ce tournant ayant causé des controverses, le comité responsable des R^{n}RS a décidé d'écrire deux versions du R^{7}RS, une version minimaliste à la manière du R^{5}RS, et une version plus complète. La première a été ratifiée en 2013, la seconde n'a pas encore vu le jour.

Ma sémantique implémente bien sûr un sous-ensemble strict du langage (quel que soit le standard choisi), au vu de l'ampleur de celui-ci. L'accent a été mis sur l'implémentation correcte des macros hygiéniques, qui est relativement complexe.

II. Utilisation

Pour compiler la sémantique, lancer `scripts/build`. Pour exécuter un fichier Scheme, lancer `scripts/kscheme fichier.scm`.

La sémantique est testée avec :

```
$ krun --version
K version:    v5.6.90
Build date:   Thu May 11 22:42:38 CEST 2023
```

III. Aperçu des fonctionnalités implémentées

Cette section passe en revue les différents aspects du langage qui sont implémentés.

III. 1. Types de données

Les valeurs peuvent être de l'un des types de données prédéfinis suivants :

- Nombres entiers et flottants.
- Booléens (notés `#t` et `#f` en Scheme).
- Chaînes de caractères.
- Symboles.
- Liste vide `()` et paires (donc indirectement listes).
- Valeur spéciale `*unspecified*`, renvoyée lorsque les standards disent que la valeur de retour d'une fonction est non-spécifiée (la convention d'utiliser une valeur nommée « `*unspecified*` » est inspirée de l'implémentation Guile).
- Cellules mutables (*boxes*). Ces objets contiennent simplement une référence à un autre objet, qui peut être modifiée. Ils sont notamment utilisés pour implémenter les environnements.
- Identificateurs, ou symboles enrichis (voir la section sur les macros hygiéniques).
- Procédure prédéfinie ou procédure anonyme (fonction lambda).

III. 2. Syntaxe lexicale et grammaire des programmes

Scheme est connu pour être un langage homoïconique, ce qui signifie qu'il possède la même syntaxe pour les données et les programmes. Une *représentation externe* d'une valeur est une chaîne de caractères qui est interprétée, dans la syntaxe lexicale de Scheme, en cette valeur. Un programme Scheme est une représentation externe d'une valeur Scheme d'une forme particulière.

Les représentations externes prennent les formes suivantes.

- Les nombres sont écrits en notation décimale (avec un point pour les décimaux). Exemples : `5`, `-4.3`
- Les chaînes de caractères sont entre guillemets et peuvent contenir quelques chaînes d'échappement comme `\n`. Exemple : `"Hello\nWorld"`. (J'ai simplement utilisé la syntaxe `String` de K, sans chercher à implémenter les chaînes d'échappement exactes spécifiées dans les standards.)
- Les booléens sont `#t` (*true*) et `#f` (*false*).
- La liste vide est `()`. Une paire de `A` et `B` est notée `(A . B)`. De plus, l'absence de point est un raccourci pour une liste. Ainsi, `(A B C)` est équivalent à `(A . (B . (C . ())))` et `(A B . C)` est équivalent à `(A . (B . C))`.
- Les symboles sont des suites de caractères d'une liste assez large qui comprend notamment les lettres, les chiffres (sauf pour le premier caractère), et de nombreux caractères spéciaux comme `! ? < > - + * /`. (Encore une fois, je n'ai pas cherché à être parfaitement conforme au standard sur ce point, d'autant que K ne prend pas en charge l'Unicode.)
- Les syntaxes « spéciales » `'x` et `#'x` sont des raccourcis pour `(quote x)` et `(syntax x)`.

Un programme prend l'une des formes suivantes.

- Constante : nombre, chaîne de caractères, booléen. Ces objets s'évaluent à eux-mêmes.
- Symbole : il s'évalue à une variable.
- Liste : appel de fonction ou de macro. Le premier élément est la fonction ou la macro, les autres sont les arguments. (La liste vide `()` n'est pas un programme valide en soi, mais peut être passée à des macros comme `quote`.)

III. 3. Macros fondamentales disponibles

- La macro fondamentale `lambda` crée une fonction anonyme. Elle est analogue à l'opérateur λ du λ-calcul. Elle a la syntaxe :

  ```
  (lambda (paramètre1 paramètre2 ...)
    expression1
    expression2
    ...)
  ```

  À l'exécution, les expressions sont évaluées dans l'ordre et la valeur de la dernière est renvoyée. Les précédentes peuvent avoir des effets de bord.

  Il existe également la forme alternative

  ```
  (lambda arguments
    expression1
    expression2
    ...)
  ```

  Dans ce cas, la fonction accepte un nombre variable d'arguments, et à l'exécution, la liste de tous les arguments est mise dans la variable `arguments`.

  Enfin, il existe une forme mixte :

  ```
  (lambda (paramètre1 paramètre2 ... . reste)
    expression1
    expression2
    ...)
  ```

La fonction accepte un certain nombre de paramètres obligatoires, et des paramètres arbitraires à la fin, mis dans une liste.

- La macro `define` définit une variable globale. Sa syntaxe est

  ```
  (define nom valeur)
  ```

  ou

  ```
  (define (nom paramètres)
    expression1
    expression2
    ...)
  ```

  Cette dernière forme est un sucre syntaxique pour

  ```
  (define nom
    (lambda (paramètres)
      expression1
      expression2
      ...))
  ```

  Dans les deux cas, la définition est autorisée à être récursive.

- La macro `set!` remplace la valeur associée à une variable par une autre valeur. (Cela est complètement différent d'une mutation de la valeur associée à la variable, qui existe aussi ; voir `set-car!`, `set-cdr!` et `set-box!`.)

- La macro `if` crée une expression conditionnelle. Elle prend la forme `(if condition expression-oui expression-non)` ou `(if condition expression-oui)`. L'évaluation de `expression-oui` et `expression-non` est paresseuse. Toutes les valeurs sauf `#f` sont considérées comme vraies. Dans le cas `(if condition expression-oui)`, si `condition` s'évalue à `#f`, la valeur renvoyée est `*unspecified*`.

- La macro `quote` insère une expression littérale. L'évaluation de l'expression `quote` renvoie simplement l'expression en argument.

- Les macros `define-syntax`, `syntax` et `syntax-case` servent à définir des macros ; voir plus bas.

III. 4. Autres macros

- `(begin expr1 expr2 ...)` évalue les expressions `expr1`, `expr2`, …, et renvoie la valeur de la dernière.

- `(and condition1 condition2 ...)` et `(or condition1 condition2 ...)` sont le ET et le OU logiques classiques. Ils sont paresseux.

- `(let ((var1 val1) (var2 val2) ...) expr1 expr2 ...)` évalue `val1`, `val2`, …, affecte les valeurs aux variables `var1`, `var2`, …, et évalue `expr1`, `expr2`, …, dans un contexte lexical local qui contient ces variables. Cette macro peut s'exprimer en termes de `lambda`.

  Il existe la variante du « let nommé », qui s'écrit `(let nom ((var1 val1) (var2 val2) ...) expr1 expr2 ...)`. Lors de l'évaluation de `expr1`, `expr2`, …, elle définit dans l'environnement lexical non seulement les variables, mais aussi une fonction `nom` qui peut être appelée sur de nouvelles valeurs pour exécuter à nouveau les mêmes expressions. Ainsi, le classique

  ```
  let rec loop arg1 arg2 ... =
    ...
  in loop init1 init2 ...
  ```

  en OCaml s'écrit en Scheme

```
(let loop ((arg1 init1)
           (arg2 init2)
           ...)
  ...)
```

- `let*` est une variante de `let` où `var1` est définie lors de l'évaluation de `val2`, et ainsi de suite. (Il n'y a pas de « let* nommé ».)
- `letrec` et `letrec*` sont des variantes où *toutes* les variables sont visibles lors de l'évaluation de *toutes* les valeurs, ce qui sert à définir des fonctions récursives, voire mutuellement récursives. La différence entre les deux est que `letrec*` doit obligatoirement évaluer les expressions dans l'ordre, tandis que `letrec` peut en théorie optimiser différemment. Dans cette sémantique, ces macros sont équivalentes.
- `cond` est un raccourci pour écrire une chaîne de `if`. Sa forme est `(cond clause1 clause2 ...)`. Chaque clause comporte une condition. Ces conditions évaluées dans l'ordre, et la première clause à avoir une condition vraie a son code exécuté.

  Les formes possibles pour les conditions sont :

  `(condition expr1 expr2 ...)`

  Évalue les expressions à la suite, renvoie la valeur de la dernière.

  `(condition)`

  Si la condition est vraie, renvoie sa valeur.

  `(condition => fonction)`

  Si la condition est vraie, applique `fonction` à sa valeur.

  `(else expr1 expr2 ...)`

  `else` est une condition toujours vraie.
- `(when condition expr1 expr2 ...)` est un raccourci pour `(if condition (begin expr1 expr2 ...))`, et `(unless condition expr1 expr2 ...)` un raccourci pour `(if (not condition) (begin expr1 expr2 ...))`.
- `(case val ((literal ...) expr1 expr2 ...))` évalue `val`, et pour chaque clause `((literal ...) expr1 expr2 ...)`, si la valeur est égale au sens de `eqv?` (voir plus bas) à l'une des valeurs de `literal ...`, évalue `expr1 expr2 ...`, sinon continue. Comme pour `cond`, la dernière clause peut aussi être `(else expr1 expr2 ...)`.
- `(rec name expr)` définit une fonction récursive `name` en évaluant `expr` dans un contexte où existe la variable `name` (équivalent OCaml : `let rec name = expr in name`).

with-syntax Voir la section sur les macros.

III. 5. Bibliothèque de fonctions prédéfinies

La plupart de ces fonctions sont définies dans les standards, mais quelques unes ne le sont pas (par exemple `display-to-string`).

Pour les nombres, les standards définissent des fonctions polymorphes qui fonctionnent à la fois sur les entiers et les flottants, mais je n'ai implémenté que des fonctions monomorphes qui fonctionnent sur les entiers *ou* sur les flottants.

Quelques fonctions existantes ne sont pas mentionnées ici lorsqu'elles servent uniquement à implémenter des fonctions de plus haut niveau. Par exemple, `primitive-display` est une fonction

interne définie dans les fichiers 𝕂, et `display` est définie dans le prélude en termes de `primitive-display` et `display-to-string`.

III. 5. a. Égalité

(eqv? x y) (*eqv* abrège *equivalent.*) Ce prédicat d'égalité se comporte globalement comme une égalité superficielle. Deux objets de types différents ne sont jamais considérés comme égaux. Les nombres, les constantes `'()` et `*unspecified*`, ainsi que les symboles, sont comparés par valeur. (Un flottant ne peut pas être considéré comme égal à un entier, les deux étant de types différents). Les chaînes de caractères et les paires sont comparés par identité.

(eq? x y) Ce prédicat a un comportement moins spécifié que `eqv?` sur les nombres et les chaînes de caractères vides. Il est fait pour être utilisé comme test d'identité. Dans cette sémantique, il est identique à `eqv?`.

(equal? x y) Prédicat d'égalité profonde (récursive).

III. 5. b. Paires

(cons x y) Paire de x et y.
(car p) | (cdr p) Premier, deuxième élément de la paire p.
(set-car! p x) | (set-cdr! p x) Mute la paire p pour changer son *car* ou *cdr* en x.

III. 5. c. Arithmétique

(exact+ x y) | (exact- x y) | (exact-neg x) | (exact* x y) Arithmétique sur les entiers (addition, soustraction, négation, multiplication).

(inexact+ x y) | (inexact- x y) | (inexact-neg x) | (inexact* x y) Idem, sur les flottants.

(exact<? x y) | (exact>? x y) | (exact<=? x y) | (exact>=? x y) Comparaisons entre entiers.

(inexact<? x y) | (inexact>? x y) | (inexact<=? x y) | (inexact>=? x y) Comparaisons entre flottants.

III. 5. d. Logique

(not x) Négation : `#t` si x vaut `#f`, ou `#f` sinon.

III. 5. e. Listes

(list x1 x2 x3 ...) Crée une liste des éléments x_1, x_2, x_3, …
(reverse lst) Renverse `lst`.
(length lst) Longueur de `lst`.
(map f lst) Applique `f` à chaque élément de `lst`, renvoie une liste des résultats.
(fold f init lst) Si `lst` est une liste des éléments x_1, x_2, …, x_n, renvoie `(f xn (... (f x2 (f x1 init))))`.

III. 5. f. Chaînes de caractères

(string-length str) Longueur de `str`.
(substring str start end) Sous-chaîne de `str`, de `start` inclus à `end` exclus.
(string-append str1 str2 ...) Concatène `str1`, `str2`, …
(number->string x) Convertit un nombre en chaîne de caractères.
(symbol->string x) Convertit un symbole en chaîne de caractères.
(string->symbol x) Convertit une chaîne de caractères en symbole.

III. 5. g. Prédicats de type

Ces fonctions renvoient `#t` si leur argument est d'un certain type et `#f` sinon.

(integer? x) | (float? x) | (boolean? x) | (symbol? x) | (null? x) | (unspecified? x) | (procedure? x) | (identifier? x) | (box? x) | (pair? x) Prédicats pour les types primitifs.

(number? x) Entier ou flottant.

(list? x) Liste vide, ou paire dont le *cdr* est une liste.

III. 5. h. Cellules (*boxes*)

Ces fonctions ne sont pas partie des standards, mais sont spécifiées dans la SRFI 111[1].

(box x) Crée une cellule initialisée à x.
(unbox b) Renvoie la valeur dans la cellule b.
(set-box! b x) Met à x la valeur dans la cellule b.

III. 5. i. Affichage

(display x) Affiche x de manière « raisonnable » sur *stdout*. Cela n'affiche *pas* une représentation externe, même si x admet des représentations externes. En particulier, une chaîne de caractères est affichée telle quelle, sans guillemets. (La fonction standard pour s'approcher d'une représentation externe est `write`, mais elle n'est pas implementée.)

(newline) Équivalent à `(display "\n")`.

(display-to-string x) Renvoie la représentation qui serait affichée par `display`, sous forme de chaîne de caractères.

III. 5. j. Macros

(free-identifier=? x y) | (bound-identifier=? x y) | (datum->syntax stx dat) | (syntax->datum stx) Voir la partie sur les macros.

III. 5. k. Autres

(read-stdin) Lit l'entrée standard et renvoie une chaîne de caractères.
(error msg) Provoque une erreur avec le message `msg`.
(assert cond) Lève une erreur si `cond` s'évalue à `#f`.

III. 6. Limitations notables

Bien sûr, il aurait été bien trop long d'implémenter la vaste bibliothèque standard spécifiée par les standards. Toutefois, on peut relever l'absence de certaines fonctionnalités que l'on pourrait (subjectivement) considérer comme « de base » mais que je n'ai pas eu le temps d'implémenter :

- Les macros `quasiquote`, `unquote` et `unquote-splicing` (ainsi que leurs cousines `quasisyntax`, `unsyntax` et `unsyntax-splicing`).
- Le type de données « caractère » (on peut toutefois représenter un caractère par une chaîne de caractères à un caractère).
- Les nombres rationnels.
- Les macros `let-syntax` et `letrec-syntax` (seule `define-syntax` est implémentée, on ne peut définir que des macros globales).

[1] https://srfi.schemers.org/srfi-111/srfi-111.html

- Les fonctions `apply` et `call/cc`.

Dans cette sémantique, `define` est restreinte à être utilisée uniquement pour des variables globales, alors que dans une implémentation plus complète, on peut aussi l'utiliser pour des variables locales (cette restriction simplifie grandement la macro-expansion).

Il n'y a pas de glaneur de cellules (*garbage collector*). Par conséquent, la mémoire allouée n'est pas libérée. Un effet malheureux de cette limitation est que la récursivité terminale (*tail recursion*) ne fonctionne pas tout à fait. La sémantique comporte une règle de récursion terminale, qui enlève bien de la pile l'environnement de la fonction appelante avant un appel terminal, mais les cellules allouées pour les variables ne sont pas libérées, si bien que la récursion n'est pas à mémoire constante.

De manière générale, les messages d'erreurs sont laconiques et n'ont pas de numéro de ligne.

La fonction `datum->syntax` a quelques problèmes liés aux variables globales (du reste, l'interaction des macros hygiéniques avec les variables globales est mal spécifiée et diffère entre implémentations, voir par exemple [1]).

IV. Fonctionnement de la sémantique

IV. 1. Configuration

La configuration choisie comporte quatre cellules.

- La cellule `<k>` contient la pile d'évaluation, comme usuellement en $\mathbb{K}$. Elle sert aussi à la pile de macro-expansion.
- La cellule `<heap>` simule la mémoire. Elle contient une table (`Map`) qui associe des adresses (représentées sous forme d'entiers) à des valeurs. Les allocations ajoutent des entrées à cette table.
- La cellule `<env>` contient une pile d'environnements d'exécution. Ceux-ci sont représentés comme des tables qui associent des numéros de variable à des cellules mutables allouées dans la mémoire (le fait d'encapsuler chaque variable dans une cellule mutable est nécessaire pour pouvoir modifier la valeur d'une variable avec `set!`). Lorsqu'une fonction est exécutée, un nouvel environnement est ajouté à la pile `<env>`, et il est retiré dès que la fonction renvoie sa valeur de retour. Les variables ne sont pas représentées par leur nom mais par un numéro, ceci en raison de l'hygiène (voir plus bas) qui peut produire des variables différentes ayant le même nom.
- La cellule `<global-env>` est la table des variables globales. Elle associe des noms de variable (`String`) à des numéros de variable (`Int`).

IV. 2. Représentation des valeurs

Dans une implémentation conventionnelle, on distingue les valeurs immédiates des valeurs non-immédiates. Ces dernières sont traditionnellement allouées sur le tas (*heap allocation*), alors que les valeurs immédiates ont une taille fixe et se passent d'allocation. Les entiers petits, les flottants, les booléens et les constantes `'()` et `*unspecified*` sont des valeurs immédiates, tandis que les chaînes de caractères, les grands entiers, les paires, les cellules et les procédures sont non-immédiates. Une valeur est alors soit une valeur immédiate, soit un pointeur vers une valeur non-immédiate.

Ma sémantique fait la même distinction, mais pour des raisons différentes des implémentations traditionnelles. En effet, en $\mathbb{K}$, contrairement au C par exemple, et comme en Scheme, il n'y a

pas de limite sur la taille d'une valeur fixée par son type, et pas de pointeurs. On pourrait donc penser à première vue que toutes les valeurs pourraient être représentées de manière uniforme, et sans pseudo-pointeurs. Cependant, cela ne permet pas de donner une sémantique correcte aux mutations. Par exemple, considérons le programme

```
(define (f p)
  (set-car! p 3))

(let ((p (cons 1 2)))
  (f p)
  (car p))
⇒ 3
```

Si la valeur `p` était passée à `f` comme une instance d'un constructeur $\mathbb{K}$, mettons `PairValue(IntValue(1), IntValue(2))`, la fonction ne pourrait pas affecter la valeur de `p` dans l'expression où elle est appelée, puisque les valeurs en $\mathbb{K}$ sont immuables. C'est pour cette raison (uniquement) que la sémantique comporte une indirection, avec un pointeur dans la cellule `<heap>`, qui modélise le tas d'allocation d'un programme C classique.

Plus précisément, la sorte `Value` est définie comme ceci dans le fichier `src/types.k` :

```
  syntax Value ::= IntValue(Int)
  syntax Value ::= FloatValue(Float)
  syntax Value ::= BoolValue(Bool)
  syntax Value ::= SymbolValue(String)
  syntax Value ::= NullValue()
  syntax Value ::= UnspecifiedValue()
  syntax Value ::= BuiltinProcedureValue(String /* name */)
  syntax Value ::= BuiltinSyntaxValue(String /* name */)
  syntax Value ::= IdentifierValue(Identifier)
  syntax Value ::= HeapValue(Int)

  syntax AllocatedValue ::= StringValue(String)
  syntax AllocatedValue ::= BoxValue(Value)
  syntax AllocatedValue ::= PairValue(Value, Value)
  syntax AllocatedValue ::= LambdaValue(Map /* environment */,
                                        LambdaArgSpec,
                                        Program /* body */)
```

Les constructeurs possibles pour `Value` sont :

- `IntValue`, `FloatValue` : valeurs numériques.
- `BoolValue` : booléen.
- `SymbolValue` : symbole.
- `NullValue` : constante `'()`.
- `UnspecifiedValue` : constante `*unspecified*`.
- `BuiltinProcedureValue` : procédure prédéfinie (distinguée par son nom).
- `BuiltinSyntaxValue` : macro prédéfinie (idem).
- `IdentifierValue` : symbole enrichi (cf. macros hygiéniques).
- `HeapValue` : ce constructeur contient une adresse (un entier), qui est une clé dans la table de la cellule `Heap`. La valeur est de sorte `AllocatedValue`.

Les constructeurs de `AllocatedValue` sont :

- `StringValue` : chaîne de caractères.
- `BoxValue` : cellule.
- `PairValue` : paire.
- `LambdaValue` : fonction lambda.

`BoxValue` et `PairValue` ne peuvent faire partie que de la sorte `AllocatedValue` à cause des mutations possibles, comme expliqué ci-dessus.

Pour `StringValue` et `LambdaValue`, la raison est différente. Les chaînes de caractères ne sont pas mutables dans cette sémantique (elles le sont dans les standards, mais je n'ai pas implémenté les opérations de mutation). En revanche, les prédicats `eqv?` et `eq?` doivent renvoyer `#t` sur des valeurs qui proviennent de la même variable. Par exemple :

```
(eqv? "abc" "abc") ⇒ non spécifié
(let ((x "abc"))
  (eqv? x x)) ⇒ #t (spécifié)
```

Ces prédicats doivent aussi s'exécuter en temps à peu près constant[2]. Pour que `(eqv? x x)` fonctionne si `StringValue` faisait partie de la sorte `Value`, la comparaison devrait être une comparaison des valeurs, caractère par caractère, donc en temps non constant et potentiellement coûteux. La même considération s'applique aux fonctions lambda, où il faudrait comparer les environnements capturés.

On peut alors se demander s'il ne serait pas plus simple de poser `syntax Value ::= Int` et de mettre tous les constructeurs dans la sorte `AllocatedValue`. La représentation des valeurs serait alors plus uniforme, et toute `Value` ne serait qu'un pointeur. C'est le choix fait par certaines implémentations. La raison du choix ci-dessus est pragmatique. Pour manipuler les objets alloués dans `<heap>`, il faut écrire des *pattern matchings* plus complexes, comme

```
<k> HeapValue(Addr) ... </k>
<heap> (Addr |-> StringValue(Str)) _Heap </heap>
```

au lieu de

```
<k> IntValue(X) ... </k>
```

De plus, à chaque fois qu'une nouvelle valeur est créée, il faut l'allouer dans `<heap>`.

Le choix de ne pas allouer les valeurs immédiates simplifie donc significativement les règles de réécriture.

IV. 3. Représentation des programmes

Les programmes sont représentés par un noyau assez petit d'opérations fondamentales. Voici les productions, toujours définies dans `src/types.k` :

```
  syntax Program ::= Value
  syntax Program ::= VarProgram(Int)
  syntax Program ::= SetProgram(Int, Program)
                     [seqstrict(2), result(Value)]
  syntax Program ::= IfProgram(Program, Program, Program)
                     [seqstrict(1), result(Value)]
  syntax Program ::= SequenceProgram(Program, Program)
                     [seqstrict(1), result(Value)]
  syntax Program ::= LambdaProgram(LambdaArgSpec, Program)
  syntax Program ::= PrimitiveDefineProgram(
                       String /* name */, Int /* var number */, Program)
  syntax Program ::= DefineSyntaxProgram(
                       String /* name */, Int /* var number */, Program)
  syntax Program ::= ApplicationProgram(Program, List /* of programs */)
                     [seqstrict(1), result(Value)]
```

[2]Le temps n'est pas tout à fait constant au moins pour `eqv?`, à cause des entiers de taille arbitraire. Néanmoins, il est rare d'avoir des entiers de milliers de chiffres, alors qu'il est très courant d'avoir des chaînes de milliers de caractères.

```
syntax Program ::= SyntaxProgram(Template)
syntax Program ::= SyntaxCaseProgram(Program, List /* of clauses */)
                   [seqstrict(1), result(Value)]
```

Nous avons donc : les programmes constants, les références à une variable, les modifications d'une variable (`set!`), les expressions conditionnelles (`if`), la séquence (`begin`), la lambda-abstraction (`lambda`), les définitions globales (`define` et `define-syntax`), l'application de fonction, et les opérations fondamentales `syntax-case` et `syntax`.

En particulier, même les expressions `let` sont compilées à l'aide de `lambda`, ce qui n'est pas efficace en pratique, mais reste simple.

IV. 4. Étapes essentielles de l'évaluation

Une première étape consiste à transformer l'arbre syntaxique donné par 𝕂 en une valeur de sorte `Value`. Cette étape est définie dans `src/read.k`. C'est par exemple là que le point au milieu de `(2 . 4)` est interprété pour donner une paire. Cette étape est assez simple.

La seconde phase est la macro-expansion de l'expression. Elle est définie dans `src/expand.k`, avec des outils auxiliaires dans `src/expand-utils.k`. Elle fait appel aux fonctions sur les symboles enrichis définies dans `src/identifier.k`. La macro-expansion des expressions `syntax-case` fait appel à `src/pattern-compile.k` pour précompiler les motifs.

Enfin, l'expression compilée en programme peut être évaluée. Cette phase est codée dans `src/eval.k`. Ses outils auxiliaires sont dans `src/eval-utils.k`. Le *pattern matching* des motifs `syntax-case` se trouve dans `src/pattern-match.k` et `src/pattern-match-utils.k`. Le remplacement des points de suspension dans les expressions `syntax` est dans `src/eval-syntax.k` et fait appel à `src/template.k`.

Le fichier `src/configuration.k` contient la configuration. `src/init.k` est chargé de calculer les valeurs initiales des cellules (d'après les macros et fonctions prédéfinies).

IV. 5. Prélude

Le prélude, qui se trouve dans le fichier `src/prelude.scm`, est une suite de définitions de fonctions et macros chargée avant l'exécution du programme. Cela permet de garder le noyau écrit en 𝕂 relativement petit, en écrivant certaines des fonctions et macros directement en Scheme.

IV. 6. Environnements

L'existence de `set!` implique que les environnements ne sont pas des associations d'une variable à une valeur, mais des associations d'une variable à une cellule mutable qui contient la valeur. Cela est nécessaire pour que les modifications se répercutent partout où l'environnement a été capturé. En effet, il serait possible lors d'un `set!` de modifier simplement la valeur associée à la variable dans l'environnement courant, ce qui en 𝕂 (puisque les valeurs sont immuables) revient à remplacer l'environnement courant par un nouvel environnement où la valeur est modifiée ; mais cela ne fonctionnerait pas pour remplacer la valeur dans les fonctions lambda qui ont capturé cet environnement.

Par exemple, cette représentation permet de faire fonctionner correctement le programme suivant :

```
(define (make-getter-and-setter initial)
  (let ((var initial))
    (cons (lambda () var)
          (lambda (new) (set! var new)))))
```

```
(let* ((getter-setter (make-getter-and-setter 0))
       (getter (car getter-setter))
       (setter (cdr getter-setter)))
  (setter 1)
  (getter))
⇒ 1
```

IV. 7. Lien entre macro-expansion et exécution

Scheme permet aux macros d'exécuter du code arbitraire. Cela pose un problème immédiat : comment définir une fonction de manière à ce que les macros puissent l'utiliser ? En effet, la définition d'une fonction normale n'est en général pas exécutée immédiatement. La macro-expansion est d'abord effectuée sur *tout* le programme, ce qui est une forme de compilation, *puis* le programme est exécuté (éventuellement à partir d'une forme précompilée).

Pour cette sémantique, cela est loin d'être une question purement théorique. En effet, nombre des fonctions fondamentales sont définies dans le prélude. Sans résoudre ce problème, il ne serait pas possible, par exemple, d'utiliser la fonction `not` dans une macro…

La solution couramment adoptée est celle d'introduire une macro, souvent appelée `begin-for-syntax` (qui n'est pas standardisée dans les R^{n}RS), qui marque les sections à exécuter dès la compilation. Ainsi, le code

```
(define (macro-helper ...)
  ...)

(define-syntax mac
  (lambda (synt)
    ... (macro-helper ...) ...))
```

ne fonctionne pas, mais peut être corrigé en

```
(begin-for-syntax
  (define (macro-helper ...)
    ...))

(define-syntax mac
  (lambda (synt)
    ... (macro-helper ...) ...))
```

Pour cette sémantique, j'ai fait le choix d'une autre solution, pour des raisons de simplicité. Au lieu d'introduire `begin-for-syntax`, j'ai modifié l'ordre normal entre macro-expansion et exécution pour que chaque expression apparaissant au plus haut niveau du programme (typiquement `define` ou `define-syntax`) soit expansée puis *directement* évaluée. En somme, on pourrait dire que la sémantique fonctionne comme un interpréteur interactif plutôt que comme un compilateur.

L'inconvénient principal est qu'il n'est pas possible de définir des fonctions mutuellement récursives à l'aide de `define`, car dans le code

```
(define (even? x)
  (or (= 0 x) (odd? (1- x))))

(define (odd? x)
  (even? (1- x)))
```

la fonction `even?` est expansée dans un environnement qui ne contient pas encore la fonction `odd?`, donc la variable `odd?` n'est pas définie à l'intérieur de `even?`.

Il reste toutefois possible de définir des fonctions mutuellement récursives, simplement il faut passer par un contournement :

```
(define even? #f)
(define odd? #f)
(set! even?
  (lambda (x)
    (or (= 0 x) (odd? (1- x)))))
(set! odd?
  (lambda (x)
    (even? (1- x))))
```

IV. 8. Macros hygiéniques

IV. 8. a. Historique et but

L'une des forces des langages de la famille Lisp est de permettre, de par leur syntaxe parenthésée et préfixe, d'étendre les possibilités syntaxiques du langage en définissant de nouvelles macros, qui sont simplement de nouvelles possibilités pour l'opérateur spécial `op` qui se trouve dans une expression `(op ...)` (ces expressions sont des appels de macro s'il existe une macro `op`, sinon des appels de fonction). Traditionnellement, les macros sont implémentées comme des fonctions particulières qui obéissent à une ordre d'évaluation différent. Pour une fonction normale, les arguments sont évalués d'abord, puis ils sont passés à la fonction, et la valeur de retour devient la valeur de l'expression entière. Pour une macro, les arguments ne sont pas évalués, mais sont passés à la macro directement (en tant qu'objets ordinaires, ce qui est rendu possible par l'homoïconicité). Puis le résultat de la macro est évalué en tant qu'expression.

En Scheme, pour permettre l'écriture de compilateurs (par opposition aux interpréteurs), l'expansion des macros se fait dans une phase séparée, avant l'exécution, et non pas dynamiquement pendant l'exécution. Le résultat de l'expansion est une représentation interne à l'implémentation, qui permet d'exécuter le programme. Le principe de ce type de macros reste le même.

Prenons un exemple « fil rouge » pour illustrer le propos. Oubliant temporairement que le langage contient la macro `or`, on souhaite la réimplementer. Pour simplifier, nous ne nous préoccuperons pas du fait que l'arité de `or` est variable et n'implémenterons qu'un `or` à deux arguments, noté `or2`. Une définition possible de cette macro avec un système de macro traditionnel est :

```
(define-macro (or2 expr1 expr2)
  `(let ((val ,expr1))
    (if val val ,expr2)))
```

Le problème rencontré avec les macros classiques est celui du manque d'hygiène lexicale : les variables introduites dans la macro peuvent masquer des variables utilisées dans les arguments, et vice-versa. Voici deux illustrations :

```
(let ((val 'foo))
  (or2 #f val))
```

Ici, l'expansion donne un programme équivalent à

```
(let ((val 'foo))
  (let ((val #f))
    (if val val val)))
```

Le dernier `val` devrait intuitivement, d'après le code source de départ, passer à `not` la valeur de la variable `val` définie à la première ligne, mais celle-ci a été masquée par celle insérée par la macro à la deuxième ligne.

Un exemple illustrant le problème inverse est :

```
(let ((if 'x))
  (or2 #t #f))
```

L'expansion donne

```
(let ((if 'x))
  (let ((val #t))
    (if val val #f)))
```

Le problème est dans le fait que le `if` inséré par la macro est lié par le `let` du programme de départ, et ne se réfère plus à la macro fondamentale `if`.

Dans l'article [2], le premier problème est appelé *introduced-binder hygiene*, et le second *introduced-reference hygiene*. (Cet article est intéressant par ailleurs, en proposant une formalisation de ce qu'est l'expansion hygiénique du point de vue logique, au-delà de son aspect opérationnel, algorithmique.)

Pour résoudre ce problème, des systèmes de macros dits hygiéniques ont été proposés. À ma connaissance, le premier article décrivant un tel système de macros est [3]. On trouve également [4] et [5], entre autres, qui proposent des algorithmes différents pour maintenir l'hygiène.

Le fonctionnement interne de ces algorithmes n'est pas sans conséquence pour l'utilisateur, car il arrive de vouloir écrire des macros volontairement non-hygiéniques. C'est le cas, par exemple, si on souhaite écrire une macro `while` à l'intérieur de laquelle appeler la fonction `break` interrompt la boucle, ce qui implique d'introduire non-hygiéniquement une variable `break` dans le code généré. Tous les algorithmes proposés permettent généralement d'écrire des macros hygiéniques de manière similaire, mais les manières d'écrire des macros non-hygiéniques peuvent différer. C'est notamment pour cette raison que le R^{5}RS ne standardise qu'un système de macro relativement simple et limité, mais entièrement hygiénique ; les implémentations sont libres de choisir un algorithme pour maintenir l'hygiène. Ce système est appelé `syntax-rules`.

IV. 8. b. Le système syntax-rules

Concrètement, les macros `syntax-rules` fonctionnent de manière semblable à des systèmes de réécriture. Une macro est définie à l'aide de `define-syntax` et créée avec `syntax-rules`. À l'intérieur de l'expression `syntax-rules` se trouvent des motifs (*patterns*), et pour chaque motif, un patron (*template*). L'expression source entière (soit `(macro argument1 argument2 ...)`, où le nom de la macro elle-même est inclus) est unifiée avec les motifs dans l'ordre, jusqu'à ce qu'un motif corresponde, et le résultat de la macro est alors le patron correspondant, où les variables liées par le motif sont remplacées par leurs valeurs. Voici comment la macro `or2` peut être définie avec `syntax-rules` :

```
(define-syntax or2
  (syntax-rules ()
    ((or2 expr1 expr2)
     (let ((val expr1))
       (if val val expr2)))))
```

Le motif `_` est, comme usuellement dans les langages avec *pattern matching*, traité comme *wildcard*.

Pour illustrer quelques autres possibilités de `syntax-rules`, voici une définition d'un `or` d'arité variable, adaptée du R^{7}RS-small :

```
(define-syntax or
  (syntax-rules ()
    ((_) #f)
    ((_ test) test)
    ((_ test1 test2 ...)
```

```
      (let ((x test1))
        (if x x (or test2 ...))))))
```

On voit qu'il est possible d'utiliser des points de suspension dans les motifs et dans les patrons. Pour le *pattern matching*, ils ont la sémantique d'une étoile de Kleene. Dans les patrons, il insèrent toutes les occurrences de la variable.

Le premier argument de la macro `syntax-rules`, qui est `()` dans les exemples précédent, est une liste de littéraux, également appelés « mot-clés ». Lorsqu'un mot-clé apparaît dans un motif, il ne peut pas s'unifier avec n'importe quelle expression, comme c'est le cas normalement d'un symbole dans un motif, mais seulement avec le mot-clé exact. L'application typique est la définition de macros comme `cond` et `case`, qui comportent des mots-clés comme `else` et `=>`. Schématiquement, une définition partielle de `cond` peut ressembler à :

```
(define-syntax cond
  (syntax-rules (else)
    ...
    ((cond (else expr1 expr2 ...))
     (begin expr1 expr2 ...))))
```

Il faut toutefois mentionner que, à nouveau par souci d'hygiène, la détection des mots-clés n'est pas une simple comparaison des symboles.

```
(cond (else 'foo)) ⇒ foo
(let ((else #f))
  (cond ((else 'foo)))) ⇒ *unspecified*
```

Le mot-clé n'est détecté que lorsqu'il est lié à la même variable que lors de la définition de la macro. Nous y reviendrons.

Avec ses règles de réécritures, points de suspension et mots-clés, le système de macros `syntax-rules` permet d'exprimer la plupart des macros courantes. Néanmoins, il a ses limites. Il est difficile (quoique pas tout à fait impossible ; cf. [6]) d'écrire des macros non-hygiéniques (comme `while` avec `break`) avec `syntax-rules`. Il n'est pas possible de générer des noms de variable, comme le font certains systèmes orientés objet avec des macros utilisées par exemple comme ceci :

```
(define-class class
  field1 field2 field3)
```

qui définirait des accesseurs `class-field1`, `class-field2` et `class-field3`.

Pour permettre l'écriture de ce type de macros, les auteurs du R^{6}RS ont standardisé une extension du système `syntax-rules`, nommée `syntax-case`, fixant ainsi, au contraire du R^{5}RS, un choix d'implémentation interne du système de macros, ce qui est plus contraignant pour les implémentations de Scheme.

IV. 8. c. Le système syntax-case

Le système `syntax-case` vise à permettre plus de flexibilité. Donnons d'abord la traduction d'une macro `syntax-rules` en `syntax-case`. Le code

```
(define-syntax macro
  (syntax-rules (<keywords>)
    ((_ <pattern>)
     <template>)
    ((_ <pattern>)
     <template>)
    ...))
```

devient

```
(define-syntax macro
  (lambda (syntax-object)
    (syntax-case syntax-object (<keywords>)
      ((_ <pattern>)
       (syntax <template>))
      ((_ <pattern>)
       (syntax <template>))
      ...)))
```

On note plusieurs modifications. L'invocation de `syntax-rules` est remplacée par une fonction d'un objet syntaxique, ce qui traduit le fait que `syntax-rules` ne fait que produire un « transformateur de syntaxe » (*syntax transformer*, soit une fonction qui prend et renvoie des objets de type « syntaxe »). La macro `syntax-case` permet le « pattern matching » sur l'objet « syntaxe », de manière similaire à `syntax-rules`. La différence principale est que les motifs sont remplacés par du code arbitraire qui génère un objet « syntaxe ». Pour utiliser les variables définies dans les motifs (*pattern variables*), il faut en conséquence passer par une syntaxe spéciale, `(syntax ...)`, qui agit de manière similaire à `(quote ...)`, mais crée un objet « syntaxe ».

Venons-en à la nature de ces objets « syntaxe ». Pour maintenir l'hygiène, il faut nécessairement distinguer les noms de variables identiques qui proviennent des macros ou du code d'origine. Pour cela, on ne peut pas utiliser les symboles (qui sont uniques en Scheme). On utilisera donc des symboles enrichis avec d'autres informations.

Au départ, cet ajout d'information est destiné à des informations permettant de préserver l'hygiène, mais il est assez naturel de l'étendre pour qu'il conserve aussi une trace du numéro de ligne où a été introduit chaque symbole, pour pouvoir donner des messages d'erreur lisibles. Comme les numéros de ligne ne s'attachent pas qu'aux symboles mais aussi à tous les autres types d'expressions, le standard laisse la liberté de choisir entre représenter toutes les expressions par des objets « syntaxe » spéciaux, ou bien représenter les expressions avec des objets normaux sauf symboles (listes, chaînes de caractères, …) et de n'attacher de l'information supplémentaire qu'aux symboles, qui deviennent des « symboles enrichis ». Pour simplifier, j'ai choisi la deuxième option (et ma sémantique n'attache pas de numéros de ligne aux symboles enrichis).

Si le système `syntax-case` est plus puissant, c'est parce qu'il permet de manipuler les objets « syntaxe ». Plusieurs primitives sont fournies à cette fin.

- `(syntax->datum x)` : Cette fonction convertit un objet « syntaxe » en enlevant l'information supplémentaire qui y est attachée. Dans cette sémantique, elle renvoie donc une copie de son argument, où tous les symboles enrichis sont remplacés par des symboles simples.

Avec cette fonction, on peut inspecter la structure des arguments de la macro. Par exemple, il devient beaucoup plus facile de savoir si un argument est une variable, avec `(symbol? (syntax->datum arg))` (cela est difficile, bien que, de manière surprenante, possible, avec `syntax-rules` ; voir [7]).

- `(datum->syntax stx d)` : Cette fonction crée un objet « syntaxe » en attachant à une expression `d` de l'information supplémentaire copiée de l'objet « syntaxe » existant `stx`.

  Cette fonction est très puissante. Elle permet de briser complètement l'hygiène lexicale. Par exemple, on peut écrire la macro `while` en introduisant dans la sortie de la macro l'objet `(datum->syntax #'while 'break)` comme variable (`#'while` est le symbole enrichi qui représente le nom de la macro ; on l'utilise ici simplement parce que c'est une partie de l'expression qui est garantie d'être un symbole enrichi). Comme il est créé avec le même

contexte syntaxique que l'invocation de la macro, il peut lier des variables de manière non-hygiénique.

- `(free-identifier=? a b)`

  Les arguments `a` et `b` sont des symboles enrichis. Cette fonction détermine s'il sont égaux « en tant que variables libres ». Voir plus bas.

  C'est la fonction qu'utilise `syntax-rules` en interne pour détecter les mots-clés.

- `(bound-identifier=? a b)`

  Comme `free-identifier=?`, mais l'égalité est considérée pour `a` et `b` « en tant que variables liées ». Voir plus bas.

  Une application de cette fonction est de vérifier qu'il n'y a pas de doublons dans les variables liées par une macro comme `let` ou `lambda`.

IV. 9. Algorithme de préservation de l'hygiène

Cette section explique brièvement comment `syntax-case` et les fonctions associées parviennent à maintenir l'hygiène lexicale.

Le R[6]RS donne une description opérationnelle plus détaillée que le R[5]RS ou le R[7]RS, mais qui reste assez succinte et ne permet pas de comprendre tous les détails. Je me suis appuyé sur [4], une référence citée dans le R[6]RS, pour comprendre l'algorithme, ainsi que sur [2].

En particulier, [2] donne un exemple assez éclairant que voici :

```
(define x 3)

(define-syntax let-inc
  (syntax-rules ()
    ((let-inc u v)
     (let ((u (exact+ 1 u)))
       v))))

(define-syntax mac
  (syntax-rules ()
    ((mac y)
     (let-inc x (exact* x y)))))

(mac x) ⇒ 12
```

L'expansion hygiénique donne

```
(define x1 3)
(let ((x2 (exact+ 1 x1)))
  (exact* x2 x1))
```

L'intérêt de cet exemple est dans le traitement du symbole enrichi `x` introduit par la macro `mac` dans l'expression `(let-inc x (exact* x y))`. D'un côté, ce symbole doit être distingué d'une manière ou d'une autre du symbole `x` de `(mac x)` car ce dernier n'est pas affecté par un `let` qui lie le premier. On pourrait penser qu'il suffirait que les `x` de `(let-inc x (exact* x y))` soient un symbole unique fraîchement généré. Pourtant, l'expansion de `(let-inc x (exact* x y))` produit aussi une référence à la variable `x` qui provient du premier `x` de `(let-inc x (exact* x y))`. En d'autres termes, cet exemple illustre la nécessité de conserver *deux* informations à la fois dans un symbole enrichi, à savoir la variable à laquelle il se réfère (un numéro attaché à l'expansion d'un `let`, `define` ou `lambda`), mais aussi un numéro qui indique quelles variables il peut lier lui-même. On a donc une « partie de référence » et une « partie de liaison ». La «

partie de référence » indique à quelle variable le symbole est lié. Cette variable change au cours de la macro-expansion au fur et à mesure que des macros comme `let` apparaissent. La valeur finale détermine la liaison dans le programme. Quant à la « partie de liaison », elle permet de garder en mémoire quelles variables peuvent se lier entre elles si elles venaient à apparaître l'une comme variable liée par un `let` et l'autre dans le corps du `let`.

- Lorsque la macro-expansion rencontre une variable, elle est simplement remplacée par sa partie de référence.
- Pour une macro qui lie une variable, comme `let`, elle crée une « partie de référence » fraîche, et elle parcourt l'expression à la recherche de variables ayant la même « partie de liaison » que la variable qui devient liée. Les variables trouvées voient leur « partie de référence » changée en la nouvelle « partie de référence » fraîche.
- En recevant le résultat de l'application d'une macro, il faut changer les parties introduites par la macro pour remplacer les « parties de liaison » par des valeurs fraîches.

L'algorithme de [4] est une généralisation de cette idée. Le problème avec cet algorithme tel quel est qu'il ne permet pas d'implémenter `datum->syntax`. En effet, une fois que la partie de référence ou la partie de liaison est remplacée, l'historique des modifications est perdu. Or, cet historique doit être conservé sur un symbole enrichi `x`, car si `(datum->syntax x 'foo)` est utilisé, le résultat doit se voir appliquer les mêmes modifications. L'algorithme introduit donc deux opérations possibles sur les symboles enrichis : la substitution, et le marquage. On a trois constructeurs pour un symbole enrichi :

```
SymboleEnrichi ::=
  SymboleSimple
  | Substitution(SymboleEnrichi, SymboleEnrichi, NuméroVariable)
  | Marque(SymboleEnrichi, Marque)
```

Les macros comme `let` opèrent des substitutions (`symbole-enrichi → Substitution(symbole-enrichi, variable-liée, numéro-frais)`). Lorsqu'une macro renvoie une expression, les symboles introduits sont marqués avec une marque fraîche. Cela nécessite de reconnaître les symboles introduits (les distinguer de ceux qui étaient déjà dans les arguments de la macro), ce qui est réalisé en appliquant une « anti-marque » aux arguments avant de les passer à la macro.

On peut alors définir `free-identifier=?` et `bound-identifier=?` en termes de la partie de référence et la partie de liaison. Deux symboles enrichis sont `free-identifier=?` si et seulement si leurs parties de référence respectives sont égales. Pour `bound-identifier=?`, hélas, il existe deux définitions possibles, toutes les deux sensées, et les articles et standards cités n'utilisent pas tous la même. La première demande que les parties de liaison soient égales, la deuxième que les parties de liaison *et les parties de référence* soient les mêmes. Cela conduit à des différences observables. Voir https://lists.gnu.org/archive/html/guile-user/2023-07/msg00018.html pour une comparaison. (J'ai choisi la deuxième option.)

L'article [4] décrit aussi un algorithme modifié qui est équivalent mais a une meilleure complexité. Pour simplifier l'implémentation déjà complexe, j'ai choisi l'algorithme plus naïf.

V. Tests

Le dossier `tests` contient une batterie importante de tests.

Dans `tests/builtins/` se trouvent des tests systématiques de chaque élément du langage implémenté (macro ou fonction).

Le dossier `tests/macros/` contient quelques tests spécifiques à l'hygiène. Néanmoins, ce ne sont pas les seuls tests des macros (il y a également `tests/builtins/syntax.scm`, `tests/builtins/syntax-case.scm`, `tests/builtins/syntax-rules.scm`, `tests/builtins/syntax->datum.scm` et `tests/builtins/datum->syntax.scm`).

Enfin, le dossier `tests/unlambda/` contient un test un peu plus élaboré qui montre les possibilités d'écrire de vrais programmes avec cette sémantique. Il s'agit d'un interpréteur du langage de programmation ésotérique Unlambda inventé par David Madore (voir [8]), fondé sur le système de combinateurs SKI. Plus précisément, j'ai porté l'interpréteur Unlambda écrit par David Madore pour qu'il n'utilise que des fonctionnalités prises en charge par KScheme (j'ai donc notamment remplacé les caractères par des chaînes de caractères à un élément). Les programmes `.unl` proviennent du même site et peuvent être exécutés avec

```
scripts/kscheme tests/unlambda/unlambda.scm < tests/unlambda/programme.unl
```

Références

[1] O. Kiselyov, "A dark, under-specified corner of R5RS macros". Accessed: Jul. 29, 2023. [Online]. Available: https://okmij.org/ftp/Scheme/macros.html#syntax-rule-dark-corner

[2] M. D. Adams, "Towards the Essence of Hygiene". 2015.

[3] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba, "Hygienic Macro Expansion". 1986.

[4] R. K. Dybvig, "Syntactic Abstraction in Scheme". 1992.

[5] M. Flatt, "Bindings as Sets of Scopes", *POPL*. 2016.

[6] O. Kiselyov, "How to write Seemingly Unhygienic and Referentially Opaque Macros with syntax-rules". 2003. Accessed: Jul. 29, 2023. [Online]. Available: https://okmij.org/ftp/Scheme/macros.html#dirty-macros

[7] O. Kiselyov, "How to write 'symbol?' with 'syntax-rules'". Accessed: Jul. 29, 2023. [Online]. Available: https://okmij.org/ftp/Scheme/macros.html#macro-symbol-p

[8] D. A. Madore, "The Unlambda Programming Language". Accessed: Jul. 29, 2023. [Online]. Available: http://www.madore.org/~david/programs/unlambda/